Bx 2015 — Bidirectional Transformations

Proceedings of the 4th International Workshop on Bidirectional Transformations

Cunha, Alcino ; Kindler, Ekkart

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Alcino Cunha, Ekkart Kindler (Eds.)

Bidirectional Transformations

4th International Workshop, Bx 2015
L'Aquila, Italy
July 24, 2015
Proceedings

Preface

This is the proceedings of the *4th International Workshop on Bidirectional Transformations* (Bx 2015). Bidirectional transformations (Bx) are a mechanism for maintaining the consistency of at least two related sources of information. Such sources can be relational databases, software models and code, or any other document following standard or ad-hoc formats. Bx are an emerging topic in a wide range of research areas, with prominent presence at top conferences in several different fields (namely databases, programming languages, software engineering, and graph transformation), but with results in one field often getting limited exposure in the others. Bx 2015 was a dedicated venue for Bx in all relevant fields and part of a workshop series that was created in order to promote cross-disciplinary research and awareness in the area. As such, since its beginning in 2012, the workshop rotated between venues in different fields. In 2015, Bx was co-located with STAF for the first time, and was previously held at the following locations:

1. Bx 2012: Tallinn, Estonia, co-located with ETAPS

2. Bx 2013: Rome, Italy, co-located with ETAPS

3. Bx 2014: Athens, Greece, co-located with EDBT/ICDT

The call for papers attracted 11 complete submissions (14 abstracts were initially submitted) from which the program committee, after a careful reviewing and discussion process, selected 7 papers for presentation at the workshop (6 regular papers and 1 tool paper):

- Michael Johnson and Robert Rosebrugh: *Spans of Delta Lenses*

- Faris Abou-Saleh, James McKinna, and Jeremy Gibbons: *Coalgebraic Aspects of Bidirectional Computation*

- Michael Johnson and Robert Rosebrugh: *Distributing Commas, and the Monad of Anchored Spans*

- Zirun Zhu, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu: *BiYacc: Roll Your Parser and Pretty-Printer into One* (tool paper)

- Soichiro Hidaka, Martin Billes, Quang Minh Tran, and Kazutaka Matsuda: *Trace-based Approach to Editability and Correspondence Analysis for Bidirectional Graph Transformations*

- James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens: *Towards a Principle of Least Surprise for Bidirectional Transformations*

- Anthony Anjorin, Erhan Leblebici, Roland Kluge, Andy Schürr, and Perdita Stevens: *A Systematic Approach and Guidelines to Developing a Triple Graph Grammar*

In addition to the presentation of these papers, the program of Bx 2015 consisted of two panel discussions. The first one, focussing on "Benchmarks and reproducibility", addressed topics such as: the current status and evolution perspectives of the Bx Examples Repository; how to best support the reproduction of paper results; or how to replicate in this community successful benchmarking initiatives from other areas. The second panel, focussing on "Reaching out to end-users", tried to identify what would be necessary for Bx languages and tools to be more applied in practice, and addressed questions such as: should we just invest more time in making existing tools more stable, usable, and better documented? or do we still need to improve the underlying Bx techniques to provide stronger guarantees to end users, namely some sort of least change or "least surprise"? We hope these panels helped the Bx community take an interest in aspects of Bx that must be improved for its research to have a real impact in different application fields. These might also pave the way for interesting submissions to next year's Bx workshop, which will be held on April 8th, 2016, in Eindhoven, The Netherlands, again co-located with ETAPS.

We would like to thank the Program Committee and the external reviewers for their detailed reviews and careful discussions, and for the overall efficiency that enabled the tight schedule for reviewing. We would also like to thank all the authors and participants for helping us make Bx 2015 a success.

June 2015,
Alcino Cunha (INESC TEC and Universidade do Minho) and
Ekkart Kindler (Technical University of Denmark, DTU)
PC chairs of Bx 2015

Program Committee

- Anthony Anjorin, Chalmers | University of Technology
- Anthony Cleve, University of Namur
- Alcino Cunha (co-chair), INESC TEC and Universidade do Minho
- Romina Eramo, University of L'Aquila
- Jeremy Gibbons, University of Oxford
- Holger Giese, Hasso Plattner Institute at the University of Potsdam
- Soichiro Hidaka, National Institute of Informatics
- Michael Johnson, Macquarie University
- Ekkart Kindler (co-chair), Technical University of Denmark (DTU)
- Peter McBrien, Imperial College London
- Hugo Pacheco, INESC TEC and Universidade do Minho
- Jorge Perez, Universidad de Chile
- Arend Rensink, University of Twente
- Perdita Stevens, University of Edinburgh
- James Terwilliger, Microsoft Corporation
- Meng Wang, University of Kent
- Jens Weber, University of Victoria
- Yingfei Xiong, Peking University

External Reviewers

- Dominique Blouin
- Johannes Dyck
- Nuno Macedo
- James McKinna
- Uwe Wolter

Table of Contents

Spans of Delta Lenses

Michael Johnson
Departments of Mathematics and Computing, Macquarie University
Robert Rosebrugh
Department of Mathematics and Computer Science
Mount Allison University

Abstract

As part of an ongoing project to unify the treatment of symmetric lenses (of various kinds) as equivalence classes of spans of asymmetric lenses (of corresponding kinds) we relate the symmetric delta lenses of Diskin et al, with spans of asymmetric delta lenses. Because delta lenses are based on state spaces which are categories rather than sets there is further structure that needs to be accounted for and one of the main findings in this paper is that the required equivalence relation among spans is compatible with, but coarser than, the one expected. The main result is an isomorphism of categories between a category whose morphisms are equivalence classes of symmetric delta lenses (here called fb-lenses) and the category of spans of delta lenses modulo the new equivalence.

1 Introduction

In their 2011 POPL paper [4] Hoffmann, Pierce and Wagner defined and studied (set-based) symmetric lenses. Since then, with the study of variants of asymmetric lenses (set-based or otherwise), there has been a need for more definitions of corresponding symmetric variants. This paper is part of an ongoing project by the authors to develop a unified theory of symmetric and asymmetric lenses of various kinds. The goal is to make it straightforward to define the symmetric version of any, possibly new, asymmetric lens variant (and conversely). Once an asymmetric lens is defined the unified theory should provide the symmetric version (and vice-versa).

In [4] Hoffmann et al noted that there were two approaches that they could take to defining symmetric lenses. One involved studying various *right* and *left* (corresponding to what other authors call *forwards* and *backwards*) operations. The other would be based on spans of asymmetric lenses. In both cases an equivalence relation was needed to define composition of symmetric lenses, to ensure that that composition is associative, and to identify lenses which were equivalent in their updating actions although they might differ in "hidden" details such as their *complements* (see [4]) or the head (peak) of their spans. Hoffmann et al gave their definition of symmetric lens in terms of left and right update operations, noting that "in the span presentation there does not seem to be a natural and easy-to-use candidate for ... equivalence".

In [6] the present authors developed the foundations needed to work with spans of lenses of various kinds and proposed an equivalence for spans of well-behaved set-based asymmetric lenses (called here HPW-lenses) and also for several other set-based variants. Our goal was to find the finest equivalence among spans of HPW lenses that would satisfy the requirements of the preceding paragraph. Such an equivalence needed to include, and be coarser than, span equivalence (an isomorphism between the heads of the spans commuting with the

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

legs of the spans). Furthermore pre-composing the legs of a span of HPW lenses with a non-trivial HPW lens gives a new span which differs from the first only in the "hidden" details — the head would be different but the updating actions at the extremities would be the same — so such pairs of spans should also be equivalent. In [6] we were able to show that the equivalence generated by such non-trivial HPW lenses (commuting with the legs of the spans) worked well, and that result was very satisfying because it demonstrated, as so much work in the theory of lenses does, that lenses and bidirectional transformations more generally are valuable generalisations of isomorphisms.

Of course, the work so far, being entirely set-based, is still far from a unified theory, so in this paper we turn to the category-based delta-lenses of Diskin, Xiong and Czarnecki [1] and study the symmetric version derived from spans of such lenses and compare it with the symmetric (forwards and backwards style) version that Diskin et al propose in [2].

The paper is structured as follows. In Sections 2 and 3 we review and develop the basic mathematical properties of delta-lenses (based on [1] and referred to here as d-lenses) and symmetric delta lenses (based on [2], and called here fb-lenses after their basic operations called "forwards" and "backwards", thus avoiding clashing with the general use of "symmetric" for equivalence reduced spans of lenses). As in the work of Hoffmann et al, both fb-lenses and spans of d-lenses need to be studied modulo an equivalence relation and the two equivalence relations we propose are introduced in Section 4. In Section 5 we show that using the two equivalences does indeed yield a category of (equivalence classes of) fb-lenses and a category of (equivalence classes of) spans of d-lenses respectively (and of course, we need to show that the equivalence relations we have introduced are congruences in order to construct the categories). Finally in Section 6 we explore the relationship between the two categories and show that there is an equivalence of categories, indeed in this case an isomorphism of categories, between them.

Because of the usefulness of category-based lenses (in particular delta-lenses) in applications, the work presented here lays important mathematical foundations. Furthermore the extra mathematical structure provided in the category-based variants has revealed a surprise — an equivalence generated by non-trivial lenses is not coarse enough to ensure that two spans of d-lenses with the same fb-behaviour are always identified. The difficulty that arises is illustrated in a short example and amounts to "twisting" the structures so that no single lens can commute with the lenses on the left side of the spans, and at the same time commute with the lenses on the right side of the spans. The solution, presented as one of the equivalences in Section 4, relaxes the requirement that the comparison be itself a lens, and asks that it properly respect the put operations on both sides (rather than having its own put operation commuting with both sides).

2 Asymmetric delta lenses

For any category $\mathbf{C}$, we write $|\mathbf{C}|$ for the set (discrete category) of objects of $\mathbf{C}$ and $\mathbf{C^2}$ for the category whose objects are arrows of $\mathbf{C}$. For a functor $G : \mathbf{S} \longrightarrow \mathbf{V}$, denote the "comma" category, whose objects are pairs consisting of an object S of $\mathbf{S}$ and an arrow $\alpha : GS \longrightarrow V$, by $(G, 1_{\mathbf{V}})$. We recall the definition of a delta lens (or d-lens) [1, 5]:

Definition 1 *A (very well-behaved)* delta lens (d-lens) *from* $\mathbf{S}$ *to* $\mathbf{V}$ *is a pair* (G, P) *where* $G : \mathbf{S} \longrightarrow \mathbf{V}$ *is a functor (the "Get") and* $P : |(G, 1_{\mathbf{V}})| \longrightarrow |\mathbf{S^2}|$ *is a function (the "Put") and the data satisfy:*

(i) d-PutInc: the domain of $P(S, \alpha : GS \longrightarrow V)$ *is* S

(ii) d-PutId: $P(S, 1_{GS} : GS \longrightarrow GS) = 1_S$

(iii) d-PutGet: $GP(S, \alpha : GS \longrightarrow V) = \alpha$

(iv) d-PutPut: $P(S, \beta\alpha : GS \longrightarrow V \longrightarrow V') = P(S', \beta : GS' \longrightarrow V')P(S, \alpha : GS \longrightarrow V)$ *where* S' *is the codomain of* $P(S, \alpha : GS \longrightarrow V)$

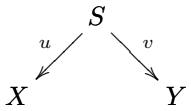
For examples of d-lenses, we refer the reader to [1]. Meanwhile, we offer a few sentences here to help orient the reader to the notations used. Both the categories $\mathbf{S}$ and $\mathbf{V}$ represent state spaces. Objects of $\mathbf{S}$ are states and an arrow $S \longrightarrow S'$ of $\mathbf{S}$ represents a specific transition from the state S which is its domain to the state S' which is its codomain. Such specified transitions are often called "deltas". Similarly for $\mathbf{V}$. A functor $G : \mathbf{S} \longrightarrow \mathbf{V}$ maps states of $\mathbf{S}$ to states of $\mathbf{V}$ — S is sent to GS. Furthermore, being a functor it maps deltas $S \longrightarrow S'$ in $\mathbf{S}$ to deltas $GS \longrightarrow GS'$ in $\mathbf{V}$. The objects of $(G, 1_{\mathbf{V}})$ are important because, being a pair $(S, \alpha : GS \longrightarrow V)$, they encapsulate

both an object of $\mathbf{S}$ and a delta starting at GS. Such a pair is the basic input for a Put operation. The Put operation P itself, in the case of a d-lens, is just a function (not a functor) and it takes such an "anchored delta" $(S, \alpha : GS \longrightarrow\!\!\!\!\rightarrow V)$ in $\mathbf{V}$ to a delta in $\mathbf{S}$ which, by d-PutInc, starts at S. The axioms d-PutId and d-PutPut ensure that the P operation respects composition, including identities. Finally, the axiom d-PutGet ensures that, as expected, the Put operation results in a delta in $\mathbf{S}$ which is carried by G to α, the given input delta.

In [5], we proved that d-lenses compose, that a c-lens as defined there is a special case of d-lens and that their composition is as for d-lenses, and finally that d-lenses are strictly more general than c-lenses. We also proved that d-lenses are certain algebras for a semi-monad.

Furthermore, in [6] we developed the technique to compose spans of lenses in general. For d-lenses, this specializes to Definition 3. We first need a small but important proposition, and we formally remind the reader about our notations for spans and cospans.

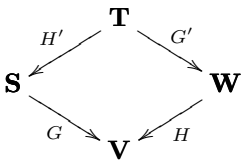
A *span* is a pair of morphisms, with common domain:



Despite the symmetry, such a span is often described as a "span from X to Y", and is distinguished from the same two arrows viewed as a span from Y to X. The illustrated span above is often denoted for brevity's sake $u : X \longleftarrow S \longrightarrow\!\!\!\!\rightarrow Y : v$ and, when X, S and Y are understood or easily derived, we sometimes just refer to it as the span u, v. The object S is sometimes called the *head* or *peak* of the span and the arrows u and v are called the *legs* of the span. The objects X and Y are, naturally enough, called the *feet* of the span. Cospans are described and notated in the same way but the arrows u and v are reversed. Finally, if, as sometimes is necessary, a span is drawn upside down, the common domain is still called the head despite being drawn below the feet.

When working with spans it is often necessary to calculate pullbacks. For simplicity of presentation we will usually assume that the pullback has been chosen so that its objects and morphisms are pairs of objects from the categories from which it has been constructed (so, for example, in the diagram below, objects of $\mathbf{T}$ are pairs of objects (S, W) from the categories $\mathbf{S}$ and $\mathbf{W}$ respectively, with the property that $G(S) = H(W)$, and similarly for morphisms of $\mathbf{T}$).

Proposition 2 *Let $G : \mathbf{S} \longrightarrow\!\!\!\!\rightarrow \mathbf{V} \longleftarrow \mathbf{W} : H$ be a cospan of functors. Suppose that $P : |(G, 1_{\mathbf{V}})| \longrightarrow\!\!\!\!\rightarrow |\mathbf{S^2}|$ is a function which, with G, makes the pair (G, P) a d-lens. Then, in the pullback square in* **cat***:*

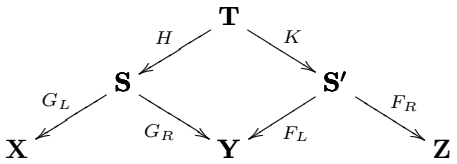


the functor G' together with $P' : |(G', 1_{\mathbf{W}})| \longrightarrow\!\!\!\!\rightarrow |\mathbf{T^2}|$ defined by $P'((S, W), \beta : G'(S, W) \longrightarrow\!\!\!\!\rightarrow W') = (P(S, H(\beta)), \beta) : (S, W) \longrightarrow\!\!\!\!\rightarrow (S', W')$ define a d-lens from $\mathbf{T}$ to $\mathbf{W}$.

Proof. Note first that P' makes sense since $H(\beta)$ is a morphism $HG'(S, W) \longrightarrow\!\!\!\!\rightarrow H(W')$ but $HG'(S, W) = GH'(S, W) = GS$ so it is in fact a morphism $GS \longrightarrow\!\!\!\!\rightarrow H(W')$. Furthermore we denote the codomain of $P(S, H(\beta))$ by S' so that $G(S') = H(W')$ and thus (S', W') is an object of $\mathbf{T}$. The d-PutInc, d-PutId and d-PutGet conditions on (G', P') are satisfied by construction. The d-PutPut condition follows immediately from d-PutPut for (G, P). ∎

This means we can talk about the "pullback" of a d-lens along an arbitrary functor, in particular along the Get of another d-lens. This is similar to the situations described in [6]. The inverted commas around "pullback" are deliberate because the constructed d-lens may not be an actual pullback in the category of d-lenses (the category whose objects are categories and whose arrows $\mathbf{S} \longrightarrow\!\!\!\!\rightarrow \mathbf{V}$ are d-lenses from $\mathbf{S}$ to $\mathbf{V}$).

Definition 3 *Suppose that in*



the functors G_L, G_R, F_L, and F_R are the Gets of d-lenses with corresponding Puts P_L, P_R, Q_L, and Q_R, and $\mathbf{T}$ *is the pullback of G_R and F_L. For the "pullback" d-lenses with Gets H and K, denote the Puts by P_H and P_K. Then the* span composite *of the span of d-lenses (G_L, P_L), (G_R, P_R) from* $\mathbf{X}$ *to* $\mathbf{Y}$ *with the span of d-lenses (F_L, Q_L), (F_R, Q_R) from* $\mathbf{Y}$ *to* $\mathbf{Z}$*, denoted*

$$((G_L, P_L), (G_R, P_R)) \circ ((F_L, Q_L), (F_R, Q_R))$$

is the span of d-lenses from $\mathbf{X}$ *to* $\mathbf{Z}$ *specified as follows. The Gets are $G_L H$ and $F_R K$. The Puts are those for the composite d-lenses $(G_L, P_L)(H, P_H)$ and $(F_R, Q_R)(K, P_K)$.*

In a sense, the composite just defined corresponds to the ordinary composite of spans in a category with pullbacks. In the category of categories, the ordinary span composition of the span G_L, G_R and with the span F_L, F_R is the span $G_L H, F_R K$. As usual for such composites, the operation is not associative without introducing an equivalence relation and we do so later in this paper.

3 Symmetric delta lenses

A symmetric delta lens (called an "fb-lens" below) is between categories, say $\mathbf{X}$ and $\mathbf{Y}$. It consists of a set of synchronizing "corrs", so named because the make explicit intended correspondences between objects of $\mathbf{X}$ and objects of $\mathbf{Y}$, together with "propagation" operations. In the forward direction, given objects X and Y synchronized by a corr r and an arrow x with domain X, the propagation returns an arrow y with domain Y and a corr synchronizing the codomains of x and y. This is made precise in the following definition and is based on definitions in [2] and [3]. We denote the domain and codomain of an arrow x by $d_0(x), d_1(x)$.

Definition 4 *Let* $\mathbf{X}$ *and* $\mathbf{Y}$ *be categories. An* fb-lens *from* $\mathbf{X}$ *to* $\mathbf{Y}$ *is $M = (\delta_{\mathbf{X}}, \delta_{\mathbf{Y}}, \mathsf{f}, \mathsf{b}) : \mathbf{X} \longleftrightarrow \mathbf{Y}$ specified as follows. The data $\delta_{\mathbf{X}}, \delta_{\mathbf{Y}}$ are a span of sets*

$$\delta_{\mathbf{X}} : |\mathbf{X}| \leftarrow R_{\mathbf{XY}} \rightarrow |\mathbf{Y}| : \delta_{\mathbf{Y}}$$

An element r of $R_{\mathbf{XY}}$ is called a corr. *For r in $R_{\mathbf{XY}}$, if $\delta_{\mathbf{X}}(r) = X, \delta_{\mathbf{Y}}(r) = Y$ the corr is denoted $r : X \leftrightarrow Y$. The data* f *and* b *are operations called* forward *and* backward propagation*:*

$$\mathsf{f} : Arr(\mathbf{X}) \times_{|\mathbf{X}|} R_{\mathbf{XY}} \rightarrow Arr(\mathbf{Y}) \times_{|\mathbf{Y}|} R_{\mathbf{XY}}$$

$$\mathsf{b} : Arr(\mathbf{Y}) \times_{|\mathbf{Y}|} R_{\mathbf{XY}} \rightarrow Arr(\mathbf{X}) \times_{|\mathbf{X}|} R_{\mathbf{XY}}$$

where the pullbacks (also known as fibered products) mean that if $\mathsf{f}(x, r) = (y, r')$, we have $d_0(x) = \delta_{\mathbf{X}}(r), d_1(y) = \delta_{\mathbf{Y}}(r')$ and similarly for b*. We also require that $d_0(y) = \delta_{\mathbf{Y}}(r)$ and $\delta_{\mathbf{X}}(r') = d_1(x)$, and the similar equations for* b*.*

Furthermore, we require that both propagations respect both the identities and composition in $\mathbf{X}$ *and* $\mathbf{Y}$*, so that we have:*

$$r : X \leftrightarrow Y \quad \textit{implies} \quad \mathsf{f}(\mathrm{id}_X, r) = (\mathrm{id}_Y, r) \quad \textit{and} \quad \mathsf{b}(\mathrm{id}_Y, r) = (\mathrm{id}_X, r)$$

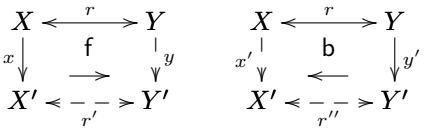
and

$$\mathsf{f}(x, r) = (y, r') \textit{ and } \mathsf{f}(x', r') = (y', r'') \quad \textit{imply} \quad \mathsf{f}(x'x, r) = (y'y, r'')$$

and

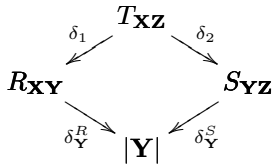
$$\mathsf{b}(y, r) = (x, r') \textit{ and } \mathsf{b}(y', r') = (x', r'') \quad \textit{imply} \quad \mathsf{b}(y'y, r) = (x'x, r'')$$

If $\mathsf{f}(x,r) = (y,r')$ and $\mathsf{b}(y',r) = (x',r'')$, we display instances of the propagation operations as:



For examples of fb-lenses we refer the reader to [2].

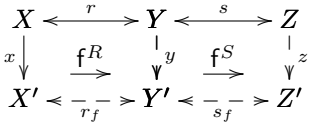
Definition 5 *Let $M = (\delta^R_{\mathbf{X}}, \delta^R_{\mathbf{Y}}, \mathsf{f}^R, \mathsf{b}^R)$ and $M' = (\delta^S_{\mathbf{Y}}, \delta^S_{\mathbf{Z}}, \mathsf{f}^S, \mathsf{b}^S)$ be two fb-lenses. We define the* composite fb-lens *$M'M = (\delta_{\mathbf{X}}, \delta_{\mathbf{Z}}, \mathsf{f}, \mathsf{b})$ as follows. Let $T_{\mathbf{XZ}}$ be the pullback of categories in*



Let $\delta_{\mathbf{X}} = \delta^R_{\mathbf{X}}\delta_1 : T_{\mathbf{XZ}} \longrightarrow \mathbf{X}$ and $\delta_{\mathbf{Z}} = \delta^S_{\mathbf{Z}}\delta_2$. The operations for $M'M$ are defined as follows. Denote $\mathsf{f}^R(x,r) = (y, r_f), \mathsf{f}^S(y,s) = (z, s_f)$ and $\mathsf{b}^S(z,s') = (y, s_b), \mathsf{b}^R(y,r') = (x, r_b)$. Then

$$\mathsf{f}(x,(r,s)) = (z,(r_f,s_f)) \text{ and } \mathsf{b}(z,(r',s')) = (x,(r_b,s_b))$$

The diagram



shows that the arities are correct for f in the forward direction. That is, we have

$$\mathsf{f} : Arr(\mathbf{X}) \times_{|\mathbf{X}|} T_{\mathbf{XZ}} \longrightarrow Arr(\mathbf{Z}) \times_{|\mathbf{Z}|} T_{\mathbf{XZ}}$$

and similarly

$$\mathsf{b} : Arr(\mathbf{Z}) \times_{|\mathbf{Z}|} T_{\mathbf{XZ}} \longrightarrow Arr(\mathbf{X}) \times_{|\mathbf{X}|} T_{\mathbf{XZ}}$$

It is easy to show that the f and b just defined respect composition and identities in $\mathbf{X}$ and $\mathbf{Z}$ and we record:

Proposition 6 *The composite $M'M$ just defined is an fb-lens from $\mathbf{X}$ to $\mathbf{Z}$.*

We note that because it is defined using a pullback, this construction of the composite of a pair of fb-lenses is not associative, and when we later define a category of fb-lenses the arrows will be equivalence classes of fb-lenses.

Next we define two constructions relating spans of d-lenses with fb-lenses.

We first consider a span of d-lenses. Let $L = (G_L, P_L)$ where $G_L : \mathbf{S} \longrightarrow \mathbf{V}$ and $K = (G_K, P_K)$ where $G_K : \mathbf{S} \longrightarrow \mathbf{W}$ be (a span of) d-lenses.

Construct the fb-lens $M_{L,K} = (\delta_{\mathbf{V}}, \delta_{\mathbf{W}}, \mathsf{f}, \mathsf{b})$ as follows:

- the corrs are $R_{\mathbf{V},\mathbf{W}} = |\mathbf{S}|$ with $\delta_{\mathbf{V}}S = G_L S$ and $\delta_{\mathbf{W}}S = G_K S$;
- forward propagation f for $v : V \longrightarrow V'$ and $S : V \leftrightarrow W$ is defined by $\mathsf{f}(v,S) = (w,S')$ where $w = G_K(P_L(S,v))$ and S' is the codomain of $P_L(S,v)$;
- backward propagation b is defined analogously.

Lemma 7 *$M_{L,K}$ is an fb-lens.*

Proof. Identity and compositionality for $M_{L,K}$ follow from functoriality of the Gets for L and K and the *d-PutId* and *d-PutPut* equations in Definition 1. ∎

In the other direction, suppose that $M = (\delta_{\mathbf{V}}, \delta_{\mathbf{W}}, \mathsf{f}, \mathsf{b})$ is an fb-lens from $\mathbf{V}$ to $\mathbf{W}$ with

$$\delta_{\mathbf{V}} : |\mathbf{V}| \leftarrow\!\!\prec R \rightarrow\!\!\succ |\mathbf{W}| : \delta_{\mathbf{W}}$$

We now construct a span of d-lenses $L_M : \mathbf{V} \leftarrow\!\!\prec \mathbf{S} \rightarrow\!\!\succ \mathbf{W} : K_M$ from $\mathbf{V}$ to $\mathbf{W}$. The first step is to define the head $\mathbf{S}$ of the span. The set of objects of $\mathbf{S}$ is the set R of corrs of M. The morphisms of $\mathbf{S}$ are defined as follows: For objects r and r', $\mathbf{S}(r, r') = \{(v, w) \mid d_0 v = \delta_{\mathbf{V}}(r), d_1 v = \delta_{\mathbf{V}}(r'), d_0 w = \delta_{\mathbf{W}}(r), d_1 v = \delta_{\mathbf{W}}(r')\}$ (where we write, as usual, $\mathbf{S}(r, r')$ for the set of arrows of $\mathbf{S}$ from r to r'). Thus an arrow may be thought of as a formal square:

$$\begin{array}{ccc} V & \overset{r}{\prec\!\!\rightarrow} & W \\ {\scriptstyle v}\downarrow & & \downarrow{\scriptstyle w} \\ V' & \underset{r'}{\prec\!\!\rightarrow} & W' \end{array}$$

Composition is inherited from composition in $\mathbf{V}$ and $\mathbf{W}$ at boundaries, or more precisely, for $(v, w) \in \mathbf{S}(r, r')$ and $(v', w') \in \mathbf{S}(r', r'')$ we define:

$$(v', w')(v, w) = (v'v, w'w)$$

in $\mathbf{S}(r, r'')$. The identities are pairs of identities. It is easy to see that $\mathbf{S}$ is a category.

Next we define the d-lens L_M to be the pair (G_L, P_L) where we define $G_L : \mathbf{S} \rightarrow\!\!\succ \mathbf{V}$ on objects by $\delta_{\mathbf{V}}$, and on arrows by projection, that is $G_L(v, w) = v$. The Put for L_M, $P_L : |(G_L, 1_{\mathbf{V}})| \rightarrow\!\!\succ |\mathbf{S^2}|$, is defined on objects $(r, v : G_L(r) \rightarrow\!\!\succ V')$ of the category $(G_L, 1_{\mathbf{V}})$ by $P_L(r, v) = (v, \pi_0 \mathsf{f}(v, r))$ which is indeed an arrow of $\mathbf{S}$ from r to $\pi_1 \mathsf{f}(v, r)$. (As is usual practice, we write π_0 and π_1 for the projection from any pair onto its first and second factors respectively.) We define $K_M = (G_K, P_K)$ similarly.

Lemma 8 *$L_M = (G_L, P_L)$ and $K_M = (G_K, P_K)$ is a span of d-lenses.*

Proof. G_L and G_K are evidently functorial. We need to show that P_L and P_K satisfy (i)-(iv) of Definition 1. These follow immediately from the properties of of the fb-lens M. ∎

The two constructions above are related. One composite of the constructions is actually the identity.

Proposition 9 *For any fb-lens M, with the notation of the constructions above*

$$M = M_{L_M, K_M}$$

Proof. By inspection, the corrs and δ's of M_{L_M,K_M} are those of M. Further, it is easy to see that, for example, the forward propagation of M_{L_M,K_M} is identical to that of M. ∎

However, the other composite of the constructions above, namely the span of d-lenses $L_{M_{L.K}}, K_{M_{L.K}}$ is not equal to the original span L, K (because the arrows of the original $\mathbf{S}$ have been replaced by the formal squares described above). We have yet to consider the appropriate equivalence for spans of d-lenses, and we do so now. We will see that $L_{M_{L.K}}, K_{M_{L.K}}$ is indeed equivalent to L, K.

4 Two equivalence relations

Our first equivalence relation is on spans of d-lenses from $\mathbf{X}$ to $\mathbf{Y}$.

Suppose that

$$\mathbf{X} \xleftarrow{(G_L, P_L)} \mathbf{S} \xrightarrow{(G_R, P_R)} \mathbf{Y} \quad \text{and} \quad \mathbf{X} \xleftarrow{(G'_L, P'_L)} \mathbf{S}' \xrightarrow{(G'_R, P'_R)} \mathbf{Y}$$

are such spans.

The functor $\Phi : \mathbf{S} \rightarrow\!\!\succ \mathbf{S}'$ is said to satisfy conditions (E) if:
(1) $G'_L \Phi = G_L$ and $G'_R \Phi = G_R$

(2) Φ is surjective on objects

(3) whenever $\Phi S = S'$, we have both $P'_L(S', G'_L S' \xrightarrow{\alpha} X) = \Phi P_L(S, G_L S \xrightarrow{\alpha} X)$ and

$$P'_R(S', G'_R S' \xrightarrow{\beta} Y) = \Phi P_R(S, G_R S \xrightarrow{\beta} Y).$$

By (1) any such Φ is a "2-cell" in spans of categories between the two $\mathbf{X}$ to $\mathbf{Y}$ spans G_L, G_R and G'_L, G'_R. Moreover, Φ is required both to be surjective on objects and also to satisfy (3), a condition which expresses a compatibility with the Puts. Notice that the identity functor on $\mathbf{S}$ satisfies conditions (E).

Definition 10 *Let $\equiv_{Sp}$ be the equivalence relation on spans of d-lenses from $\mathbf{X}$ to $\mathbf{Y}$ which is generated by functors Φ satisfying (E).*

To simplify describing $\equiv_{Sp}$ we now prove some properties of functors satisfying conditions (E).

Lemma 11 *A composite of d-lens span morphisms satisfying (E) also satisfies (E).*

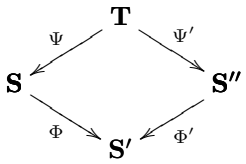
Proof. Suppose that we have spans of d-lenses $(G_L, P_L), (G_R, P_R)$ and $(G'_L, P'_L), (G'_R, P'_R)$ as above, and a third such span is:

$$\mathbf{X} \xleftarrow{(G''_L, P''_L)} \mathbf{S}'' \xrightarrow{(G''_R, P''_R)} \mathbf{Y}$$

Suppose $\Phi : \mathbf{S} \longrightarrow \mathbf{S}'$ and $\Phi' : \mathbf{S}' \longrightarrow \mathbf{S}''$ satisfy (E). Properties (1) and (2) for $\Phi'\Phi$ are immediate. We show the P'_L part of property (3) for $\Phi'\Phi$. Suppose $\Phi'\Phi S = S''$ and consider $P''_L(S'', G''_L S' \xrightarrow{\alpha} X)$. By (E) for Φ and Φ', since $\Phi'(\Phi(S)) = S''$, we have $P''_L(S'', G''_L S'' \xrightarrow{\alpha} X) = \Phi' P'_L(\Phi(S)$, and $G'_L \Phi(S) \xrightarrow{\alpha} X) = \Phi'\Phi P_L(S, G_L S \xrightarrow{\alpha} X)$ as required. ∎

Suppose that Φ satisfies (E). When $\Phi S = S'$ it follows that $G_L S = G'_L \Phi S = G'_L S'$, which we will use below. Note that if Φ were the Get of a d-lens (although it need not be) then it would be surjective on arrows by the *d-PutGet* equation, but not necessarily surjective on hom sets.

Lemma 12 *Suppose once again that $(G_L, P_L), (G_R, P_R)$, $(G'_L, P'_L), (G'_R, P'_R)$ and $(G''_L, P''_L), (G''_R, P''_R)$ are spans of d-lenses as above. Let $\Phi : \mathbf{S} \longrightarrow \mathbf{S}' \longleftarrow \mathbf{S}'' : \Phi'$ be the functors in a cospan of span morphisms satisfying (E). Let*



be a pullback in **cat**. *Then there is a span of d-lenses $\mathbf{X} \xleftarrow{(G^T_L, P^T_L)} \mathbf{T} \xrightarrow{(G^T_R, P^T_R)} \mathbf{Y}$ defined by $G^T_L = G_L \Psi$ and*

$$P^T_L((S, S''), G^T_L(S, S'') \xrightarrow{\alpha} X) = (S \xrightarrow{P_L(S,\alpha)} W, S'' \xrightarrow{P''_L(S'',\alpha)} W'')$$

and similarly for (G^T_R, P^T_R). Moreover, Ψ and Ψ' satisfy (E).

Proof. The first point is that (G^T_L, P^T_L) and (G^T_R, P^T_R) actually are d-lenses. We need to know that P^T_L is well-defined. Since (S, S'') is an object of the pullback $\mathbf{T}$, we know that $\Phi(S) = \Phi'(S'') = S'$, say. We want $P^T_L((S, S''), G^T_L(S, S'') \xrightarrow{\alpha} X)$ to be an arrow of $\mathbf{T}$, so we need to show that $\Phi(S \xrightarrow{P_L(S,\alpha)} W)$ is equal to $\Phi'(S'' \xrightarrow{P''_L(S'',\alpha)} W'')$. However both are equal to $P'_L(S', \alpha)$ since both Φ and Φ' satisfy (E). Thus furthermore, (W, W'') is an object of $\mathbf{T}$ and using this for *d-PutPut* each of the required d-lens equations is easy to establish.

Next, we show that Ψ and Ψ' satisfy (E). First of all, the Gets commute by definition. Moreover, both Ψ and Ψ' are surjective on objects because Φ and Φ' are so.

It remains to check property (3) for Ψ and Ψ'. We need to show that whenever $\Psi(S, S'') = S$, we have

$$P_L(S, G_L S \xrightarrow{\alpha} U) = \Psi(P_L^T((S, S''), G_L^T(S, S'') \xrightarrow{\alpha} U))$$

and this follows immediately from the definitions of Ψ and P_L^T. (Notice that for $S = \Psi(S, S'')$ we have $G_L S = G'_L \Phi S = G'_L \Phi'(S'') = G''_L(S'')$ and thus $P''_L(S'', G''_L S'' \xrightarrow{\alpha} U)$ is well-defined.) Similarly Ψ' satisfies (3). ∎

Corollary 13 *Zig-zags of span morphisms satisfying (E) reduce to spans of span morphisms satisfying (E).* ∎

A zig-zag is any string of arrows (ignoring the direction of the indidual arrows so that neighbouring arrows might be connected head to head or tail to tail as well as tail to head). It follows that any proof that two spans of d-lenses are $\equiv_{Sp}$ equivalent can be reduced to a single span Ψ, Ψ' of span morphisms satisfying (E).

The second equivalence relation we introduce is on the set of fb-lenses from $\mathbf{X}$ to $\mathbf{Y}$. Recall that Diskin et al [3] defined symmetric delta lenses (our fb-lenses), but they did not consider composing them. Like Hoffman et al [4] they would find that they need to consider equivalence classes of their symmetric delta lenses in order for the appropriate composition to be associative. Also like Hoffman et al, there is a need for an equivalence among their lenses to eliminate artificial differences. In fact, defining an equivalence to restore associativity is easy. Choosing the correct equivalence to eliminate the artificial differences is more delicate. And what do we mean by "atificial differences"? Symmetric lenses of various kinds include hidden data — the complements of Hoffmann et al and the corrs of Diskin et al are examples. The hidden data is important for checking and maintaining consistency, but different arrangements of hidden data with the same overall effect should not be counted as different symmetric lenses.

We now introduce such a relation on the set of fb-lenses from $\mathbf{X}$ to $\mathbf{Y}$.

Definition 14 *Let $L = (\delta_{\mathbf{X}}, \delta_{\mathbf{Y}}, \mathsf{f}, \mathsf{b})$ and $L' = (\delta'_{\mathbf{X}}, \delta'_{\mathbf{Y}}, \mathsf{f}', \mathsf{b}')$ be two fb-lenses (from $\mathbf{X}$ to $\mathbf{Y}$) with corrs $R_{\mathbf{XY}}, R'_{\mathbf{XY}}$. We say $L \equiv_{\mathsf{fb}} L'$ iff there is a relation σ from $R_{\mathbf{XY}}$ to $R'_{\mathbf{XY}}$ with the following properties:*

1. *σ is compatible with the δ's, i.e. $r\sigma r'$ implies $\delta_{\mathbf{X}} r = \delta'_{\mathbf{X}} r'$ and $\delta_{\mathbf{Y}} r = \delta'_{\mathbf{Y}} r'$*
2. *σ is total in both directions, i.e. for all r in $R_{\mathbf{XY}}$, there is r' in $R'_{\mathbf{XY}}$ with $r\sigma r'$ and conversely.*
3. *for all r, r', x an arrow of $\mathbf{X}$, if $r\sigma r'$ and $\delta_{\mathbf{X}} r$ is the domain of x then the first components of $\mathsf{f}(x, r)$ and $\mathsf{f}'(x, r')$ are equal and the second components are σ related, i.e. $\pi_0\mathsf{f}(x, r) = \pi_0\mathsf{f}'(x, r')$ and $\pi_1\mathsf{f}(x, r)\sigma\pi_1\mathsf{f}'(x, r')$*
4. *the corresponding condition for b, i.e. for all r, r', y an arrow of $\mathbf{Y}$, if $r\sigma r'$ and $\delta_{\mathbf{X}} r$ is the domain of x then $\pi_0\mathsf{b}(y, r) = \pi_0\mathsf{b}'(y, r')$ and $\pi_1\mathsf{b}(y, r)\sigma\pi_1\mathsf{b}'(y, r')$*

Lemma 15 *The relation $\equiv_{\mathsf{fb}}$ is an equivalence relation.*

Proof. For reflexivity take σ to be the identity relation; for symmetry take the opposite relation for σ; for transitivity, the composite relation is easily seen to satisfy conditions 1. to 4. ∎

Lemma 16 *For L, L' as in the definition above, consider a surjection $\varphi : R_{\mathbf{XY}} \longrightarrow R'_{\mathbf{XY}}$ compatible with the δ's and satisfying the following conditions:*

- *for corrs r, r', and an arrow x of $\mathbf{X}$, if $\varphi(r) = r'$ and $\delta_{\mathbf{X}} r$ is the domain of x then the first components of $\mathsf{f}(x, r)$ and $\mathsf{f}'(x, r')$ are equal and the second components are φ related, i.e. $\pi_0\mathsf{f}(x, r) = \pi_0\mathsf{f}'(x, r')$ and $\varphi(\pi_1\mathsf{f}(x, r)) = \pi_1\mathsf{f}'(x, r')$*
- *the corresponding condition for b, i.e. for all r, r', y an arrow of $\mathbf{Y}$, if $\varphi(r) = r'$ and $\delta_{\mathbf{X}} r$ is the domain of x then $\pi_0\mathsf{b}(y, r) = \pi_0\mathsf{b}'(y, r')$ and $\varphi(\pi_1\mathsf{b}(y, r)) = \pi_1\mathsf{b}'(y, r')$*

The collection of such surjections (viewed as relations) generates the equivalence relation $\equiv_{\mathsf{fb}}$.

Proof. To see that $\equiv_{\mathsf{fb}}$ is generated by such surjections, consider the span tabulating the relation σ in $\equiv_{\mathsf{fb}}$ viz.

$$R_{\mathbf{XY}} \twoheadleftarrow \sigma \twoheadrightarrow R'_{\mathbf{XY}}.$$

Notice that each leg is a surjection satisfying the conditions of the lemma. Conversely, any zig-zag of such surjections defines a relation which is easily seen to satisfy the conditions of Definition 14. ∎

Remark 17 Although $\equiv_{\mathsf{fb}}$ is generated by zig-zags of certain surjections, we have just seen that any such zig-zag can be reduced to a single span of such surjections between sets of corrs.

Proposition 18 *Suppose that $M \equiv_{\mathsf{fb}} M'$ are fb-lenses from $\mathbf{X}$ to $\mathbf{Y}$ equivalent by a generator for $\equiv_{\mathsf{fb}}$, i.e. a surjection $\varphi : R_{\mathbf{XY}} \twoheadrightarrow R'_{\mathbf{XY}}$. Then $(L_M, K_M) \equiv_{Sp} (L_{M'}, K_{M'})$ as spans of d-lenses from $\mathbf{X}$ to $\mathbf{Y}$.*

Proof. We first define $\Phi : \mathbf{S} \twoheadrightarrow \mathbf{S}'$ on objects by φ. Notice that Φ is surjective on objects since φ is a surjection. To define Φ on arrows of $\mathbf{S}$, consider an arrow

$$\begin{array}{ccc} X & \overset{r}{\longleftrightarrow} & Y \\ {\scriptstyle x}\downarrow & & \downarrow{\scriptstyle y} \\ X' & \underset{r'}{\longleftrightarrow} & Y' \end{array}$$

of $\mathbf{S}$. Its image under Φ is defined to be the arrow

$$\begin{array}{ccc} X & \overset{\varphi(r)}{\longleftrightarrow} & Y \\ {\scriptstyle x}\downarrow & & \downarrow{\scriptstyle y} \\ X' & \underset{\varphi(r')}{\longleftrightarrow} & Y' \end{array}$$

which is an arrow of $\mathbf{S}'$ since φ is compatible with the δs. This Φ is evidently functorial and commutes with the Gets. It remains to show that Φ satisfies condition (3) of (E), that is, whenever $\Phi(r) = r'$, $P'_L(r', G'_L r' \twoheadrightarrow X') = \Phi P_L(r, G_L r \twoheadrightarrow X')$ (with the similar equation holding for P'_K).

Now, when $r' = \Phi(r)$, we have

$$\begin{array}{rcll} P'_L(r', x) & = & (x, \pi_0 \mathsf{f}'(x, r')) & \\ & = & (x, \pi_0 \mathsf{f}(x, r)) & \text{see Lemma 16} \\ & = & \Phi(x, \pi_0 \mathsf{f}(x, r)) & \text{definition of } \Phi \\ & = & \Phi P_L(r, x) & \end{array}$$

as required. Similarly for P'_K. ∎

Proposition 19 *Suppose that $(L, K) \equiv_{Sp} (L', K')$ as spans of d-lenses from $\mathbf{X}$ to $\mathbf{Y}$ are made equivalent by a generator for $\equiv_{Sp}$, i.e. a functor $\Phi : \mathbf{S} \twoheadrightarrow \mathbf{S}'$ satisfying conditions (E). Then $M_{L,K} \equiv_{\mathsf{fb}} M_{L',K'}$ as fb-lenses from $\mathbf{X}$ to $\mathbf{Y}$.*

Proof. Let φ be the object function of Φ. Since Φ commutes with the Gets, φ is compatible with the δ's.

We need to show that φ satisfies the conditions in Lemma 16. Suppose $\Phi(S) = S'$ and $G_L(S)$ is the domain of x. Then

$$\begin{array}{rcl} \pi_0 \mathsf{f}(x, S) & = & G_K P_L(x, S) \\ & = & G'_K \Phi P_L(x, S) \\ & = & G'_K P'_L(x, S') \\ & = & \pi_0 \mathsf{f}'(x, S') \end{array}$$

and

$$\begin{aligned}\varphi\pi_1\mathsf{f}(x,S) &= \varphi d_1 P_L(x,S)\\ &= d_1\Phi P_L(x,S)\\ &= d_1 P'_L(x,S')\\ &= \pi_1\mathsf{f}'(x,S')\end{aligned}$$

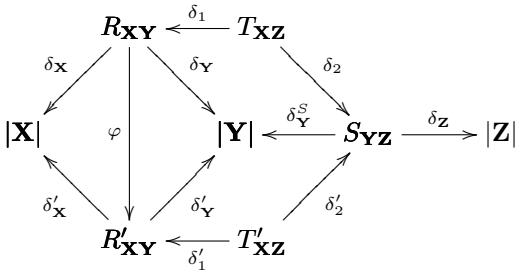
as required. Similarly for the equations involving b. ∎

5 Two categories of lenses

The collections of lenses so far discussed do not form categories since their composition is not associative. We are going to use the equivalence relations of the previous section to resolve this, but first we show that the equivalence relations respect the composites defined above, that is they are "congruences".

Proposition 20 *Suppose that $M = (\delta_{\mathbf{X}}, \delta_{\mathbf{Y}}, \mathsf{f}^R, \mathsf{b}^R)$, $M' = (\delta'_{\mathbf{X}}, \delta'_{\mathbf{Y}}, \mathsf{f}^{R'}, \mathsf{b}^{R'})$ and $N = (\delta^S_{\mathbf{Y}}, \delta_{\mathbf{Z}}, \mathsf{f}^S, \mathsf{b}^S)$ are fb-lenses (see the diagram below in which $R_{\mathbf{XY}}$, $R'_{\mathbf{XY}}$ and $S_{\mathbf{YZ}}$ are the corresponding corrs). Further, suppose $\varphi : R_{\mathbf{XY}} \longrightarrow R'_{\mathbf{XY}}$ is a generator of $\equiv_{\mathsf{fb}}$. Thus $M \equiv_{\mathsf{fb}} M'$. Then $NM \equiv_{\mathsf{fb}} NM'$.*

Proof. The composite NM has as corrs the pullback $T_{\mathbf{XZ}}$ as in Definition 5, and similarly NM' has corrs $T'_{\mathbf{XZ}}$.



In order to show that $NM \equiv_{\mathsf{fb}} NM'$, we construct $\varphi' : T_{\mathbf{XZ}} \longrightarrow T'_{\mathbf{XZ}}$. This is straightforward using the universal property of the pullback $T'_{\mathbf{XZ}}$, since $\delta'_{\mathbf{Y}}\varphi\delta_1 = \delta^S_{\mathbf{Y}}\delta_2$.

To finish, we need to check that φ' satisfies the four requirements of Definition 14.

Compatibility with δs is easy when we note that $\varphi\delta_1 = \delta'_1\varphi'$ and $\delta'_2\varphi' = \delta_2$.

The function φ' is a total relation in both directions since it is surjective. To see that φ' is surjective, note that any element of $T'_{\mathbf{XZ}}$ can be thought of as a pair (r', s) compatible over $\mathbf{Y}$, and since φ is surjective, there exists an r in $R_{\mathbf{XY}}$, necessarily compatible with s, such that $\varphi'(r, s) = (\varphi(r), s) = (r', s)$.

Let f be the forward propagation of the composite NM, as defined in Definition 5, and let f' be the forward propagation of the composite NM'. Suppose $r' = \varphi(r)$ and thus $(r', s) = \varphi'(r, s)$. We need to show that the first components of $\mathsf{f}(x, (r, s))$ and $\mathsf{f}'(x, (r', s))$ are equal and that φ' takes the second component of $\mathsf{f}(x, (r, s))$ to the second component of $\mathsf{f}'(x, (r', s))$.

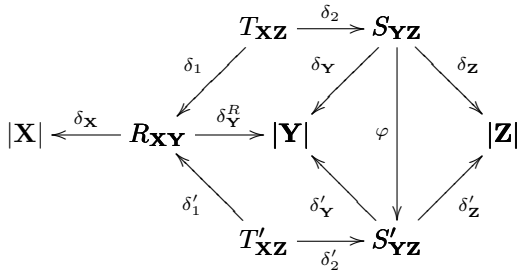
The first component of $\mathsf{f}(x, (r, s))$ is $\pi_0\mathsf{f}^S(\pi_0\mathsf{f}^R(x, r), s)$, while the first component of $\mathsf{f}'(x, (r', s))$ is $\pi_0\mathsf{f}^S(\pi_0\mathsf{f}^{R'}(x, r'), s)$, and these are equal since φ is a generator of $\equiv_{\mathsf{fb}}$ implies $\pi_0\mathsf{f}^R(x, r) = \pi_0\mathsf{f}^{R'}(x, \varphi(r))$.

The second component of $\mathsf{f}(x, (r, s))$ is $(\pi_1\mathsf{f}^R(x, r), \pi_1\mathsf{f}^S(\pi_0 f^R(x, r), s))$, while the second component of $\mathsf{f}'(x, (r', s))$ is $(\pi_1\mathsf{f}^{R'}(x, r'), \pi_1\mathsf{f}^S(\pi_0 f^{R'}(x, r'), s))$, and φ' of the first equals the second since, as before, $\pi_0\mathsf{f}^R(x, r) = \pi_0\mathsf{f}^{R'}(x, \varphi(r))$.

The same arguments work for b and b'. ∎

Proposition 21 *Suppose that $M = (\delta_{\mathbf{X}}, \delta^R_{\mathbf{Y}}, \mathsf{f}^R, \mathsf{b}^R)$, $N = (\delta_{\mathbf{Y}}, \delta_{\mathbf{Z}}, \mathsf{f}^S, \mathsf{b}^S)$ and $N' = (\delta'_{\mathbf{Y}}, \delta'_{\mathbf{Z}}, \mathsf{f}^{S'}, \mathsf{b}^{S'})$ are fb-lenses (see the diagram below in which $R_{\mathbf{XY}}$, $S_{\mathbf{YZ}}$ and $S'_{\mathbf{YZ}}$ are the corresponding corrs). Further, suppose $\varphi : S_{\mathbf{YZ}} \longrightarrow S'_{\mathbf{YZ}}$ is a generator of $\equiv_{\mathsf{fb}}$. Thus $N \equiv_{\mathsf{fb}} N'$. Then $NM \equiv_{\mathsf{fb}} N'M$.*

Proof. The composite NM has as corrs the pullback $T_{\mathbf{XZ}}$ as in Definition 5, and similarly $N'M$ has corrs $T'_{\mathbf{XZ}}$.



The proof follows the same argument as in the previous proposition. ∎

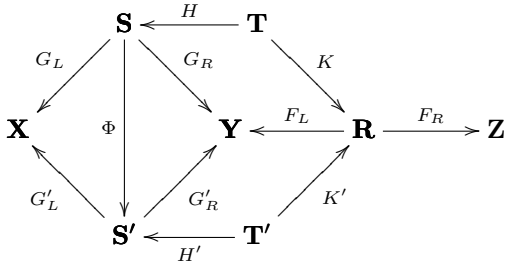
Theorem 22 *Equivalence classes for* $\equiv_{\mathsf{fb}}$ *are the arrows of a category, denoted* $\mathsf{fb}DLens$.

Proof. We first note that Propositions 20 and 21 ensure that composition is well-defined independently of choice of representative. There is an identity fb-lens with obvious structure which acts as an identity for the composition. It remains only to note that associativity follows by standard re-bracketing of iterated pullbacks. The re-bracketing function is the φ for an $\equiv_{\mathsf{fb}}$ equivalence. ∎

Proposition 23 *Suppose that* (G_L, P_L), (G_R, P_R), (G'_L, P'_L), (G'_R, P'_R), (F_L, Q_L), *and* (F_R, Q_R) *are d-lenses whose Gets are shown in the diagram below. Further, suppose* $\Phi : \mathbf{S} \longrightarrow \mathbf{S}'$ *is a functor satisfying properties (E). Thus the span* (G_L, P_L), (G_R, P_R) *is* $\equiv_{Sp}$ *to the span* (G'_L, P'_L), (G'_R, P'_R). *Then the two possible span composites are equivalent, that is*

$$((G_L, P_L), (G_R, P_R)) \circ ((F_L, Q_L), (F_R, Q_R)) \equiv_{Sp} ((G'_L, P'_L), (G'_R, P'_R)) \circ ((F_L, Q_L), (F_R, Q_R)).$$

Proof. The top composite span of d-lenses in the diagram below has head $\mathbf{T}$, the pullback of G_R and F_L (see Definition 3), similarly $\mathbf{T}'$ is the head of the bottom span composite.



In order to show the claimed equivalence, we construct a functor $\Phi' : \mathbf{T} \longrightarrow \mathbf{T}'$. Since $G'_R \Phi H = F_L K$, Φ' is defined by applying the universal property of the pullback $\mathbf{T}'$.

Since $\mathbf{T}$ and $\mathbf{T}'$ are pullbacks of functors, their objects can be taken to be pairs of objects from $\mathbf{S}$ and $\mathbf{R}$, respectively $\mathbf{S}'$ and $\mathbf{R}$. Similarly, their arrows can be taken to be pairs. Also H and K, respectively H' and K' can be taken to be projections. We can now explicitly describe the action of Φ' on an arrow of $\mathbf{T}$ as $\Phi'(t_0, t_1) = (\Phi t_0, t_1)$.

As in Definition 3, we denote the Puts of the lenses whose Gets are H and K by P_H and P_K. Similarly for H' and K'. Denote the composite lens $(G_L, P_L)(H, P_H)$ by (G, P) and similarly $(G', P') = (G'_L, P'_L)(H', P_{H'})$.

We need to show that Φ' satisfies the conditions (E). By its construction Φ' commutes with the Gets, and is surjective on objects.

It remains to show that whenever $\Phi(S, R) = (S', R')$ (which implies that $R = R'$ and $\Phi(S) = S'$) we have

$$P'((S', R'), G'_L H'(S', R') \xrightarrow{\alpha} X') = \Phi' P((S, R), G_L H(S, R) \xrightarrow{\alpha} X')$$

and

$$Q'((S',R'),F_RK'(S',R') \xrightarrow{\gamma} Z') = \Phi' Q((S,R),F_RK(S,R) \xrightarrow{\gamma} Z')$$

We begin by proving the first equation immediately above. We know that whenever $\Phi(S) = S'$, $P'_L(S', G'L(S') \xrightarrow{\alpha} X') = \Phi P_L(S, G_L(S) \xrightarrow{\alpha} X')$. We calculate

$$\begin{aligned}
P'((S',R'),G'_LH'(S',R') \xrightarrow{\alpha} X') &= P'((S',R'),G'_L(S') \xrightarrow{\alpha} X') \\
&= P_{H'}((S',R'),P'_L(S',G'_L(S') \xrightarrow{\alpha} X')) \\
&= (P'_L(S',G'_L(S') \xrightarrow{\alpha} X'),Q_L(R',G'_RP'_L(S',G'_L(S') \xrightarrow{\alpha} X'))) \\
&= (\Phi P_L(S,G_L(S) \xrightarrow{\alpha} X'),Q_L(R,G'_R\Phi P_L(S,G_L(S) \xrightarrow{\alpha} X'))) \\
&= \Phi'(P_L(S,G_L(S) \xrightarrow{\alpha} X'),Q_L(R,G_RP_L(S,G_L(S) \xrightarrow{\alpha} X'))) \\
&= \Phi'(P_H((S,R),P_L(S,G_L(S) \xrightarrow{\alpha} X'))) \\
&= \Phi' P((S,R),G_LH(S,R) \xrightarrow{\alpha} X')
\end{aligned}$$

The first step is merely that H' is a projection; the second is the definition of P' as the Put of the composite lens whose Get is G'_LH'; the third is the definition of $P_{H'}$ (see Proposition 2); the fourth uses $R' = R$ and the hypothesis stated just before the equations; the fifth follows since Φ commutes with G_R and G'_R and the definition of Φ'; the sixth is the definition of P_H (see Proposition 2); the last is the definition of P as the Put of the composite lens whose Get is G_LH.

To establish the second equation, suppose $\Phi'(S,R) = (S',R')$, whence $R = R'$ and $\Phi(S) = S'$, and so because Φ satisfies conditions (E), we have

$$P'_R(S', G'_R(S') \xrightarrow{\beta} Y') = \Phi P_R(S, G_R(S) \xrightarrow{\beta} Y')$$

and since $G_R(S) = G'_R\Phi(S) = G'_R(S')$, the right hand side can be written as $\Phi P_R(S, G'_R(S') \xrightarrow{\beta} Y')$. We calculate

$$\begin{aligned}
Q'((S',R'),F_RK'(S',R') \xrightarrow{\gamma} Z') &= P_{K'}((S',R'),Q_R(R',F_R(R') \xrightarrow{\gamma} Z')) \\
&= (P'_R(S',F_LQ_R(R',F_R(R') \xrightarrow{\gamma} Z')),Q_R(R',F_R(R') \xrightarrow{\gamma} Z'))
\end{aligned}$$

Before continuing the calculation, we simplify by defining β by $(G'_R(S') \xrightarrow{\beta} Y') = F_LQ_R(R',F_R(R') \xrightarrow{\gamma} Z') = F_LQ_R(R,F_R(R) \xrightarrow{\gamma} Z')$ after noting that $G'_R(S')$ is the domain of $F_LQ_R(R',F_R(R')\gamma Z')$ since the T' pullback square commutes. Now, continuing the calculation above:

$$\begin{aligned}
&= (P'_R(S',G'_R(S') \xrightarrow{\beta} Y',Q_R(R',F_R(R') \xrightarrow{\gamma} Z'))) \\
&= (\Phi P_R(S,G_R(S) \xrightarrow{\beta} Y',Q_R(R',F_R(R') \xrightarrow{\gamma} Z'))) \\
&= (\Phi P_R(S,G'_R(S') \xrightarrow{\beta} Y',Q_R(R,F_R(R) \xrightarrow{\gamma} Z'))) \\
&= \Phi'(P_R(S,F_LQ_R(R,F_R(R) \xrightarrow{\gamma} Z'),Q_R(R,F_R(R) \xrightarrow{\gamma} Z'))) \\
&= \Phi' P_K((S,R),Q_R(R,F_R(R) \xrightarrow{\gamma} Z'),Q_R(R,F_R(R) \xrightarrow{\gamma} Z')) \\
&= \Phi' Q((S,R),F_RK(S,R) \xrightarrow{\gamma} Z')
\end{aligned}$$

The first step uses that K' is a projection and the definition of Q' as the Put of the composite lens whose Get is F'_RK' ; the second is the definition of $P_{K'}$ (see Proposition 2); the third is the definition of β above; the fourth uses the hypothesis stated before the equations; the fifth uses $R = R'$ and the note just before the equations; the sixth uses the definitions of Φ' and β; the seventh is the definition of P_K (see Proposition 2); the last is the definition of Q as the Put of the composite lens whose Get is F_RK. ∎

Like Propositions 20 and 21, there is a reflected version of Proposition 23, showing that equivalent spans of d-lenses when composed on the left with another span of d-lenses are equivalent.

Proposition 24 *In notation analogous to Proposition 23,*

$$((G_L, P_L), (G_R, P_R)) \circ ((F_L, Q_L), (F_R, Q_R)) \equiv_{Sp} ((G_L, P_L), (G_R, P_R)) \circ ((F'_L, Q'_L), (F'_R, Q'_R)).$$

Theorem 25 *Equivalence classes for $\equiv_{Sp}$ are the arrows of a category, denoted SpDLens.*

Proof. We first note that Proposition 23 and Proposition 24 ensure that composition is well-defined independently of choice of representative. There is a span of identity d-lenses which acts as the identity for the composition. Again, associativity follows by standard re-bracketing of iterated pullbacks of categories. The re-bracketing functor is the Φ for an $\equiv_{Sp}$ equivalence. ∎

6 An isomorphism of categories of lenses

Now that we have the categories $\mathsf{fb}DLens$ and $SpDLens$, we can extend the constructions of Section 3 to functors on them.

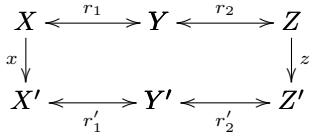
Definition 26 *For the $\equiv_{\mathsf{fb}}$ equivalence class $[M]$ of an fb-lens M, let $\mathcal{A}([M])$ be the $\equiv_{Sp}$ equivalence class of, in the notation of Lemma 8, the span L_M, K_M.*

Proposition 27 *$\mathcal{A}$ is the arrow function of a functor, also denoted $\mathcal{A}$, from $\mathsf{fb}DLens$ to $SpDLens$.*

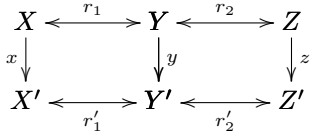
Proof. We need to show that $\mathcal{A}$ preserves identities and composition.

For the former denote by $M_{\mathbf{X}}$ the identity fb-lens on a category $\mathbf{X}$. We begin by noticing that the category $\mathbf{X}_p$ at the head of the span of d-lenses constructed from $M_{\mathbf{X}}$ has as its objects exactly those of $\mathbf{X}$. Its arrows from X to X' are arbitrary pairs of $\mathbf{X}$ arrows, both of which are from X to X'. Define the functor Φ from the head $\mathbf{X}$ of the identity span on $\mathbf{X}$ to $\mathbf{X}_p$ by sending an arrow x of $\mathbf{X}$ to the pair of arrows (x, x). This functor Φ satisfies conditions (E), and so $\mathcal{A}([M_{\mathbf{X}}]) = [\mathbf{X}]$ as required.

Let M and M' be a composable pair of fb-lenses from $\mathbf{X}$ to $\mathbf{Y}$ and $\mathbf{Y}$ to $\mathbf{Z}$ respectively. The composite fb-lens $M'M$ has as corrs compatible pairs of corrs, one from M and one from M' (see Definition 5). The head $\mathbf{S}$ of the span of d-lenses constructed from $M'M$ has objects compatible pairs of corrs and as arrows from compatible corrs (r_1, r_2) to compatible corrs (r'_1, r'_2), pairs of arrows, one from $\mathbf{X}$ and one from $\mathbf{Z}$ as shown



On the other hand, the span composite of the spans constructed from M and M' has as head a category $\mathbf{T}$ whose objects are pairs of compatible corrs from M and M' respectively. The arrows of $\mathbf{T}$ are triples of arrows (x, y, z) as shown



Define the functor Φ from $\mathbf{T}$ to $\mathbf{S}$ by sending the triple of arrows (x, y, z) to the pair of arrows (x, z). This functor Φ satisfies conditions (E), and so $\mathcal{A}([M'M]) = \mathcal{A}([M'])\mathcal{A}([M])$ as required. ∎

Definition 28 *For the $\equiv_{Sp}$ equivalence class $[L, K]$ of a span of d-lenses L, K, let $\mathcal{S}([L, K])$ be the $\equiv_{\mathsf{fb}}$ equivalence class of, in the notation of Lemma 7, the fb-lens $M_{L,K}$.*

Proposition 29 *$\mathcal{S}$ is the arrow function of a functor, also denoted $\mathcal{S}$, from $SpDLens$ to $\mathsf{fb}DLens$.*

Proof. We need to show that $\mathcal{S}$ preserves identities and composition.

Unlike the previous proof, the preservation of identities and composition is "on the nose". That is, the construction applied to the identity gives precisely the identity fb-lens. Moreover, with judicious choice of pullbacks, the construction applied to the composite of two composable spans of d-lenses *is* the composite of the fb-lenses constructed from each of the spans.

Thus $\mathcal{S}$ preserves the equivalence class of the identity span and $\mathcal{S}([L_1, K_1][L_2, K_2]) = \mathcal{S}([L_1, K_1])\mathcal{S}([L_2, K_2])$. ∎

Theorem 30 *The functors $\mathcal{A}$ and $\mathcal{S}$ are an isomorphism of categories $SpDLens \cong \mathsf{fb}DLens$.*

Proof. We need to show that the composites $\mathcal{AS}$ and $\mathcal{SA}$ are identities. Recall first that both $\mathcal{A}$ and $\mathcal{S}$ have identity functions as object functions. Considering the arrows, Proposition 9 shows that $\mathcal{SA}$ is the identity functor. We now consider $\mathcal{AS}$.

For a span L, K of d-lenses between $\mathbf{X}$ and $\mathbf{Y}$, using the notation of Lemmas 7 and 8, $\mathcal{AS}([L, K]) = [L_{M_{L,K}}, K_{M_{L,K}}]$, so we consider the span $L_{M_{L,K}}, K_{M_{L,K}}$ of d-lenses whose Gets and Puts we denote by F_L, Q_L and F_K, Q_K respectively. The head of the span is a category we denote $\mathbf{S}_{L,K}$ whose objects are the same as the objects of $\mathbf{S}$, the head of the span L, K. We define an identity on objects functor $\Phi : \mathbf{S} \longrightarrow \mathbf{S}_{L,K}$ on arrows by $\Phi(s) = (G_L(s), G_K(s))$ (recalling that arrows of $\mathbf{S}_{L,K}$ are pairs of arrows from $\mathbf{X}$ and $\mathbf{Y}$, respectively). We finish by showing that Φ satisfies conditions (E), and so witnesses $\mathcal{AS}([L, K]) = [L, K]$.

It remains to show that Φ satisfies conditions (E). Being identity on objects, Φ is certainly surjective on objects, and it commutes with the Gets by its construction. For condition (3), given an object S' of $\mathbf{S}_{L,K}$, an object S of $\mathbf{S}$ such that $\Phi S = S'$, and an arrow $\alpha : G_L(S) = F_L(S') \longrightarrow X'$ in $\mathbf{X}$, we have $P_L(S, \alpha)$ an arrow of $\mathbf{S}$. We need to show that $\Phi P_L(S, \alpha) = Q_L(S', \alpha)$. Since $\Phi S = S'$ we have $S = S'$. Now $Q_L(S', \alpha) = Q_L(S, \alpha) = (\alpha, \pi_0 \mathsf{f}(\alpha, S))$, for the forward propagation f of $M_{L,K}$ constructed as in Lemma 7. By that construction $\pi_0 \mathsf{f}(\alpha, S) = G_K(P_L(S, \alpha))$. But $\Phi P_L(S, \alpha) = (G_L(P_L(S, \alpha)), G_K(P_L(S, \alpha))) = (\alpha, \pi_0 \mathsf{f}(\alpha, S)) = Q_L(S', \alpha)$.

Thus, since $\mathcal{AS}([L, K]) = [L, K]$, $\mathcal{AS}$ is the identity. ∎

7 Conclusions

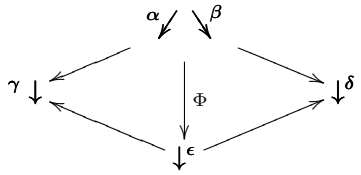
Because asymmetric delta lenses and symmetric delta lenses are so useful in applications, it is important that we understand them well and provide a firm mathematical foundation. This paper provides such a foundation by formalizing fb-lenses and their composition, including determining the equivalence required on fb-lenses for the composition to be well-defined and associative. Furthermore the resultant category $\mathsf{fb}DLens$ of fb-lenses is equivalent, indeed isomorphic, to the category $SpDLens$ of equivalence classes of spans of d-lenses.

This last result, the isomorphism between $\mathsf{fb}DLens$ and $SpDLens$, furthers the program to unify the treatment of symmetric lenses of type X as equivalence classes of spans of asymmetric lenses of type X, carrying that program for the first time into category-based lenses. (And that extension came with a surprise — see below.)

Naturally a unified treatment needs to be tested extensively on a wide range of lens types, and more work remains. The present paper is an important step in the program, and provides a reason to be hopeful that the unification is close at hand. Indeed, with this work the program encounters the important category-based lenses for the first time and that substantially widens the base of unified examples.

We end with a distilled example. It shows in a simplified way why the equivalence used here, based on conditions (E), needs to be coarser than an equivalence generated by lenses commuting with the spans though it remains compatible with the earlier work. Thus it is also a coarser equivalence relation than might have been expected based on [6].

The figure below presents two spans of d-lenses. The categories at the head and feet of the spans have been shown explicitly. In three cases the category has a single non-identify morphism called variously γ, δ and ϵ while in the fourth case the category has two distinct nonidentity morphisms denoted α and β. In all cases objects and identity morphisms have not been shown. In three cases there are just two objects, while in the fourth case there are three, with a single object serving as the domain of both α and β.



The arrows displayed in both spans represent (the Gets of) d-lenses. In the lower span the d-lenses are simply identity d-lenses (the Gets are isomorphisms sending ϵ to γ in the left hand leg, and to δ in the right hand leg). Both of the Puts are then determined. The upper span is made up of two non-identity d-lenses. In both cases the Gets send both α and β to the one non-identity morphism (γ in the left hand leg and δ in the right hand leg). We specify the Puts for the upper span (eliding reference to objects in the Puts' parameters since they can be easily deduced): $P_L(\gamma) = \alpha$ and $P_R(\delta) = \beta$ for the left and right Puts respectively.

Notice that Φ, the functor that sends both α and β to ϵ, satisfies conditions (E) showing, as expected, that the two spans are equivalent. After all, if one traces through the forward and backward behaviours across the two spans the results at the extremities are in all cases the same, though the intermediate results at the heads of the spans differ. However, Φ cannot be the Get of a lens which commutes with the other four d-lenses. Indeed, to commute with the left hand lenses would require $P_\Phi(\epsilon) = \alpha$ while to commute with the right hand lenses would require $P_\Phi(\epsilon) = \beta$, but $\alpha \neq \beta$.

8 Acknowledgements

This paper has benefited from valuable suggestions by anonymous referees. The authors are grateful for the careful and insightful refereeing. In addition, the authors acknowledge with gratitude the support of NSERC Canada and the Australian Research Council.

References

[1] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki (2011), From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case, *Journal of Object Technology* **10**, 6:1–25, doi:10.5381/jot.2011.10.1.a6

[2] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann and Francesco Orejas (2011), From State- to Delta-Based Bidirectional Model Transformations: the Symmetric Case, ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems: Springer.

[3] Diskin, Z., and T. Maibaum, Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond, 7th ACCAT Workshop on Applied and Computational Category Theory, 2012.

[4] Hofmann, M., Pierce, B., and Wagner, D. (2011) Symmetric Lenses. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas.

[5] Johnson, M. and Rosebrugh, R. (2013) Delta Lenses and Fibrations. *Electronic Communications of the EASST*, vol 57, 18pp.

[6] Johnson, M. and Rosebrugh, R. (2014) Spans of Lenses. CEUR Proceedings, vol 1133, 112–118.

Coalgebraic Aspects of Bidirectional Computation

Faris Abou-Saleh[1] James McKinna[2] Jeremy Gibbons[1]

1. Department of Computer Science
University of Oxford, UK
{Faris.Abou-Saleh,Jeremy.Gibbons}@cs.ox.ac.uk

2. School of Informatics
University of Edinburgh, UK
James.McKinna@ed.ac.uk

Abstract

We have previously (Bx, 2014; MPC, 2015) shown that several state-based bx formalisms can be captured using monadic functional programming, using the state monad together with possibly other monadic effects, giving rise to structures we have called monadic bx (mbx). In this paper, we develop a coalgebraic theory of state-based bx, and relate the resulting coalgebraic structures (cbx) to mbx. We show that cbx support a notion of composition coherent with, but conceptually simpler than, our previous mbx definition. Coalgebraic bisimulation yields a natural notion of behavioural equivalence on cbx, which respects composition, and essentially includes symmetric lens equivalence as a special case. Finally, we speculate on the applications of this coalgebraic perspective to other bx constructions and formalisms.

1 Introduction

Many scenarios in computer science involve multiple, partially overlapping, representations of the same data, such that whenever one representation is modified, the others must be updated in order to maintain consistency. Such scenarios arise for example in the context of model-driven software development, databases and string parsing [6]. Various formalisms, collectively known as *bidirectional transformations* (bx), have been developed to study them, including (a)symmetric lenses [8, 16], relational bx [30], and triple-graph grammars [27].

In recent years, there has been a drive to understand the similarities and differences between these formalisms [18]; and a few attempts have been made to give a unified treatment. In previous work [5, an extended abstract] and [1, to appear], we outlined a unified theory, with examples, of various accounts of bx in the literature, in terms of computations defined monadically using Haskell's **do**-notation. The idea is to interpret a bx between two data sources A and B (subject to some consistency relation $R \subseteq A \times B$) relative to some monad M representing computational effects, in terms of operations, $get_L : MA$ and $set_L : A \to M\ ()$, and similarly get_R, set_R for B, which allow lookups and updates on both A and B while maintaining R. We defined an *effectful bx* over a monad M in terms of these four operations, written as $t : A \Longleftrightarrow_M B$, subject to several equations in line with the Plotkin–Power equational theory of state [23]. The key difference is that in a bidirectional context, the sources A and B are interdependent, or *entangled*: updates to A (via set_L) should in general affect B, and vice versa. Thus we must abandon some of the Plotkin–Power equations; for instance, one no longer expects set_L to commute with set_R. To distinguish our earlier effectful bx from the coalgebraic treatment to be developed in this paper, we will refer to them as *monadic bx*, or simply *mbx*, from now on.

We showed that several well-known bx formalisms may be described by monadic bx for the particular case of the *state monad*, $M_S\ X = S \to (X \times S)$, called *State* S in Haskell. Moreover, we introduced a new class of bx: monadic bx with effects, expressed in terms of the *monad transformer* counterpart T_S^M to M_S (called *StateT* $S\ M$ in Haskell), where $T_S^M\ X = S \to M\ (X \times S)$ builds on some monad M, such as I/O. We defined

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

composition $t_1 \fatsemi t_2$ for those mbx with *transparent get* operations – i.e. *get*s which are effect-free and do not modify the state. The definition is in terms of $StateT$-monad morphisms derived from lenses (see Section 5), adapting work of Shkaravska [28]. As with symmetric lenses, composition can only be well-behaved up to some notion of equivalence, due to the different state-spaces involved. We showed that our definition of composition was associative and had identities up to an equivalence defined by monad morphisms – in particular, given by isomorphisms of state-spaces.

Although this equivalence is adequate for analysing such properties, it proves unsuitably fine-grained for comparing other aspects of bx behaviour. For instance, one may be interested in optimisation; one would then like some means of reasoning about different implementations, to check that they indeed have the same behaviour. Such reasoning is not possible based solely on state-space isomorphisms. As most of our work considers state-based bx, it is natural to ask whether one can identify an alternative, less ad-hoc, notion of equivalence.

In this paper, we present a coalgebraic treatment of our earlier work on monadic bx, inspired by Power and Shkaravska's work on variable arrays and comodels [24] for the theory of mutable state. Previously, we considered structure relative to the categories Set, Hask (Haskell types and functions); in particular, we argued that the composite mbx operations we defined on the underlying simple types respected appropriate set-theoretic invariants. Here, the coalgebraic approach has several benefits: firstly, it gives a more direct categorical treatment of bx composition, in terms of pullbacks. More importantly, it allows us to improve on our earlier notion of equivalence given by state-space isomorphism, appealing instead to the theory of coalgebraic bisimularity [26]; in particular, we relate it to the equivalence on symmetric lenses [16].

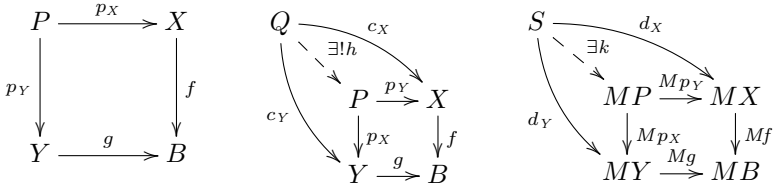
The technical contributions and structure of this paper are as follows. Firstly, in Section 2 we identify a categorical setting, and useful properties, for interpreting bx in terms of coalgebras. In Section 3, we introduce an equivalence on coalgebras, namely pointed coalgebraic bisimulation, and demonstrate how this equivalence relates to that of symmetric lenses, in addition to various effectful bx scenarios. (Pointedness identifies initial states, with respect to which coalgebra behaviours are compared.) In Section 4 we give a detailed account of composition of coalgebraic bx in terms of pullbacks, which is both more direct and categorically general than our earlier definition [1], highlighting subtleties in the definitions (such as Remark 4.8). We prove that our coalgebraic notion of composition is associative, and has identities, up to the equivalence we have introduced. Finally, in Section 5 we show that coalgebraic bx composition is coherent with that of [1].

NB We discuss only key ideas and techniques; an extended version with detailed proofs is available at `http://groups.inf.ed.ac.uk/bx/cbx-tr.pdf`.

2 Coalgebraic bx

2.1 Categorical Prerequisites

1. We will assume an ambient category $\mathbb{C}$ with finite products $X \times Y$ and exponentials X^Y (essentially, the space of functions from Y to X), on which a strong monad M is defined. (See for example Moggi's original paper [21] for a discussion of the need for such structure.) This allows us to make free use of the equational theory of **do**-notation as the internal language of the Kleisli category $\mathsf{Kl}(M)$ (see [9] for more detail). Rather than using pointfree strength and associativity isomorphisms, **do**-notation is convenient for representing value-passing between functions using pointwise syntax.
2. In order to define coalgebraic bx composition in Section 4, we further require that $\mathbb{C}$ have *pullbacks*, and that M *weakly preserve them* [11]; we say "M is **wpp**", for short. The following diagrams make this more explicit. Recall that a pullback of two arrows $f : X \to B$ and $g : Y \to B$ is an object P and span of arrows $p_X : P \to X$, $p_Y : P \to Y$ making the below-left diagram commute, such that for any object Q and span c_X, c_Y making the outermost 'square' in the middle diagram commute, there is a *unique* arrow h making the whole diagram commute. Finally, the *wpp* property (for M) asserts that the span Mp_X, Mp_Y forms a *weak* pullback of Mf and Mg: for any object S and span d_X, d_Y making the outermost 'square' in the right-hand diagram commute, there is an arrow k, *not necessarily unique*, making the whole diagram commute.



3. For technical reasons, we assume that the *return* of the monad M is monic: this allows us to observe the value x wrapped in an effect-free computation *return* x. Most computational monads (we are unaware of natural counterexamples) have this property: *e.g.* global state, I/O, non-determinism, and exceptions [22].
4. Lastly, we assume an object I, such that arrows $x : I \to X$ represent "elements" x of an object X. Typically, as in Set, I might be the terminal object 1, but if $\mathbb{C}$ is symmetric monoidal closed (*e.g.* pointed cpos and strict maps), then I could be the monoidal unit.

Remark 2.1. *The wpp condition lets us consider (at least for $\mathbb{C} = \mathsf{Set}$) monads M of computational interest such as (probabilistic) non-determinism [12, 29], which are wpp but do not preserve pullbacks; more generally, we can include I/O, exceptions, and monoid actions, by appealing to a simple criterion to check wpp holds for such M [11, Theorem 2.8].*

2.2 Bx as Pointed Coalgebras

We now give a coalgebraic description of bx, i.e. as state-based systems. We begin by noting that many bx formalisms, such as (a)symmetric lenses and relational bx, often involve an *initialised* state. The behaviours of two such bx are compared relative to their initial states. Hence, to reason about such behaviour, throughout this paper we concern ourselves with *pointed* coalgebras with designated initial state. Coalgebras with the same structure, but different initial states, are considered distinct (see [3] for more general considerations). Corollary 4.14 makes explicit the categorical structure of bx represented by such structures.

Definition 2.2. *For an endofunctor F on a category $\mathbb{C}$, a* pointed F-coalgebra *is a triple $(X, \alpha, \epsilon_\alpha)$ consisting of: an object X of $\mathbb{C}$ – the* carrier *or* state-space*; an arrow $\alpha : X \to FX$ – its* structure map *or simply* structure*; and an arrow $\epsilon_\alpha : I \to X$ picking out a distinguished initial state. We abuse notation, typically writing α for the pointed coalgebra itself.*

Now we define the behaviour functors we use to describe bx coalgebraically; as anticipated above, we incorporate a monad M into the definition from the outset to capture effects, although for many example classes, such as symmetric lenses, it suffices to take $M = Id$, the identity monad. Here are several example settings, involving more interesting monads, to which we return in Section 3.2:

- **Interactive I/O.** In [1] we gave an example of an mbx which notifies the user whenever an update occurred. Extending this example further, after an update to A or B, in order to restore consistency (as specified e.g. by a consistency relation $R \subseteq A \times B$) the bx might prompt the user for a new B or A value respectively until a consistent value is supplied. Such behaviour may be described in terms of a simplified version of the *I/O monad* $MX = \nu Y \,.\, X + (O \times Y) + Y^I$ given by a set O of observable output labels, and a set of inputs I that the user can provide. Note that the ν-fixpoint permits possibly non-terminating I/O interactions.
- **Failure.** Sometimes it may be simply impossible to propagate an update on A across to B, or vice versa; there is no way to restore consistency. In this case, the update request should simply fail; and we may model this with the *Maybe* monad $MX = 1 + X$.
- **Non-determinism.** There may be more than one way of restoring consistency between A and B after an update. In this case, rather than prompting the user at every such instance, it may be preferable for a non-deterministic choice to be made. We may model this situation by taking the monad M to be the (finitary) *powerset monad*.

Definition 2.3. *For objects A and B, define the functor $F^M_{AB}\,(-) = A \times (M\,(-))^A \times B \times (M\,(-))^B$.*

For a given choice of objects A and B, we will use the corresponding functor F^M_{AB} to characterise the behaviour of those bx between A and B, as F^M_{AB}-coalgebras. By taking projections, we can see the structure map $\alpha : X \to F^M_{AB}\ X$ of a pointed F^M_{AB}-coalgebra as supplying four pieces of data: a function $X \to A$ which observes the A-value in a state X; a function $X \to (MX)^A$ which, uncurried into the form $X \times A \to MX$, gives us the side-effectful result MX of updating the A-value in the state X; and two similar functions for B. These respectively correspond to the get_L, set_L, get_R, and set_R functions of a monadic bx with respect to the state monad M_X (with state-space X), as outlined above. Note that in this coalgebraic presentation, the behaviour functor makes clear that the *get* operations are pure functions of the coalgebra state – corresponding to 'transparent' bx [1].

Convention 2.4. *Given $\alpha : X \to F^M_{AB}\ X$, we write $\alpha.get_L : X \to A$, and $\alpha.set_L : X \to (MX)^A$, for the corresponding projections, called 'left'- or L-operations, and similarly $\alpha.get_R : X \to B$, $\alpha.set_R : X \to (MX)^B$ for the other projections, called R-operations. Where α may be inferred, and we wish to draw attention to the carrier X, we also write $x.get_L$ for $\alpha.get_L\ x$, and similarly for the other L-, R-operations.*

Remark 2.5. *Note that F^M_{AB} may be defined in terms of the costore comonad $S \times (-)^S$, used in [24] to describe arrays of independent variables. However, we differ in not allowing simultaneous update of each source (which would take $S = A \times B$), because updates to A may affect B, and vice versa.*

To ensure that pointed F^M_{AB}-coalgebras provide sensible implementations of reading and writing to A and B, we impose laws restricting their behaviour. We call such well-behaved coalgebras *coalgebraic bx*, or cbx.

Definition 2.6. *A* coalgebraic bx *is a pointed F^M_{AB}-coalgebra $(X, \alpha : X \to F^M_{AB}\ X, \epsilon_\alpha)$ for which the following laws hold at L (writing $x.get_L$ for $\alpha.get_L\ x$, etc. as per Convention 2.4):*

$$(\mathsf{CGetSet}_L)\ (\alpha): \quad x.set_L\ (x.get_L) = return\ x$$
$$(\mathsf{CSetGet}_L)\ (\alpha): \quad \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', x'.get_L)\} = \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', a)\}$$

and the corresponding laws $(\mathsf{CSetGet}_R)$ and $(\mathsf{CGetSet}_R)$ hold at R.

These laws are the analogues of the (GS), (SG) *laws [1] which generalise those for well-behaved lenses [8, see also Section 5.2 below]. They also correspond to a subset of the laws for coalgebras of the costore comonad $S \times (-)^S$, again for $S = A$ and $S = B$ independently, but excluding the analogue of 'Put-Put' or very-well-behavedness of lenses [10]. We typically refer to a cbx by its structure map, and simply write $\alpha : A \nrightarrow_X B$, where we may further omit explicit mention of X.*

Here is a simple example of a cbx, which will provide identities for cbx composition as defined in Section 4 below. Note that there is a separate identity bx for each *pair* of an object A and initial value $e : A$.

Example 2.7. *Given $(A, e : A)$, there is a trivial cbx structure $\iota(e) : A \nrightarrow_A A$ defined by $\epsilon_{\iota(e)} = e$; $a.get_L = a = a.get_R$; $a.set_L\ a' = return\ a' = a.set_R\ a'$.*

Remark 2.8. *Our definition does not make explicit any consistency relation $R \subseteq A \times B$ on the observable A and B values; however, one obtains such a relation from the get functions applied to all possible states, viz. $R = \{(a, b) : \exists x\,.\, get_L\ x = a \wedge get_R\ x = b\}$. One may then show that well-behaved cbx do, indeed, maintain consistency wrt R.*

3 Behavioural Equivalence and Bisimulation

In this section, we introduce the notion of pointed coalgebraic bisimulation, which defines a behavioural equivalence $\equiv$ for pointed cbx. In Section 3.1 we compare this equivalence to the established notion of equivalence for symmetric lenses. We then discuss in Section 3.2 the behavioural equivalences induced for the classes of effectful bx described in Section 2.2: interactive I/O, failure, and non-determinism.

Definition 3.1. *A* pointed (F-)coalgebra morphism *h between pointed coalgebras $(X, \alpha, \epsilon_\alpha)$ and $(Y, \beta, \epsilon_\beta)$ is a map $h : X \to Y$ such that $\beta \circ h = Fh \circ \alpha$ and $h \circ \epsilon_\alpha = \epsilon_\beta$.*

Remark 3.2. *In terms of* **do** *notation, $h : X \to Y$ being an F^M_{AB}-coalgebra morphism between α and β is equivalent to the following laws (where we again write $x.set_L$ for $\alpha.set_L\ x$, and so on):*

$$(\mathsf{CGetP}_L)(h): \quad x.get_L = (h\ x).get_L$$
$$(\mathsf{CSetP}_L)(h): \quad \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (h\ x')\} = \mathbf{do}\ \{\mathbf{let}\ y = (h\ x); y' \leftarrow y.set_L\ a; return\ y'\}$$

We now define a modest generalisation to $\mathbb{C}$ of the standard Set-based definition of (coalgebraic) bisimulation relations [19]. (Since we are concerned only with the *existence* of bisimulations between X and Y, we may consider them to be given non-uniquely by some jointly monic pair, as follows).

Definition 3.3. *A* bisimulation between pointed coalgebras α and β *is a pointed coalgebra ζ and a span (p, q) of pointed coalgebra morphisms $p : \zeta \to \alpha$, $q : \zeta \to \beta$ which is* jointly monic *(in $\mathbb{C}$) – meaning that whenever $p \circ f = p \circ f'$ and $q \circ f = q \circ f'$, we have $f = f'$.*

Definition 3.3 characterises bisimulation in the concrete case $\mathbb{C} = \mathsf{Set}$ and $M = Id$ as follows:

Proposition 3.4. *A pointed F^{Id}_{AB}-bisimulation R on a pair of coalgebraic bxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ implies, for all $a : A$ and $b : B$,*

- *$x.get_L = y.get_L$ and $x.get_R = y.get_R$;*
- *$(x.set_L\ a, y.set_L\ a) \in R$, and $(x.set_R\ b, y.set_R\ b) \in R$.*

Definition 3.5. *We say that two cbx α, α' are* behaviourally equivalent*, and write $\alpha \equiv \alpha'$, if there exists a pointed coalgebraic bisimulation (ζ, p, q) between α and α'.*

3.1 Relationship with Symmetric Lens Equivalence

In this subsection, we describe symmetric lenses (SL) [16] in terms of cbx, and relate pointed bisimilarity between cbx and symmetric lens (SL-)equivalence [*ibid.*, Definition 3.2]. First of all, it is straightforward to describe a symmetric lens between A and B with complement C – given by a pair of functions $putr :: A \times C \to B \times C$, $putl :: B \times C \to A \times C$ and initial state ϵ_C together satisfying two laws – as a cbx: take $M = Id$ and state-space $X = A \times C \times B$, encapsulating the current value of the lens complement C, as well as those of A and B (*cf.* [5, Section 4]). We now define the analogues of the SL-operations for a cbx between A and B:

Definition 3.6. *(Note that this is the opposite L-R convention from [16]!)*

$$x.put_L : A \to (B \times X) \qquad x.put_L\ a = \textbf{let}\ x' = x.set_L\ a\ \textbf{in}\ (x'.get_R, x')$$
$$x.put_R : B \to (A \times X) \qquad x.put_R\ b = \textbf{let}\ x' = x.set_R\ b\ \textbf{in}\ (x'.get_L, x')$$

Proposition 3.7. *Taking $\mathbb{C} = \mathsf{Set}$ and $M = Id$, a pointed F^{Id}_{AB}-bisimulation R between cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ implies the following:*

- *$x.get_L = y.get_L$ and $x.get_R = y.get_R$;*
- *for all $a : A$, $x.put_L\ a = (b', x')$ and $y.put_L\ a = (b', y')$ for some b' and $(x', y') \in R$;*
- *for all $b : B$, $x.put_R\ b = (a', x')$ and $y.put_R\ b = (a', y')$ for some a' and $(x', y') \in R$.*

Expressing cbx equivalence in these terms reveals that it is effectively identical to SL-equivalence. The cosmetic difference is that we are able to observe the 'current' values of A and B in any given state, via the *get* functions. This information is also implicitly present in SL-equivalence, where a sequence of *putr* or *putl* operations amounts to a sequence of *set*s to A and B, but where we cannot observe which values have been set. Here, the *get* operations make this information explicit. We say more about the categorical relationship between cbx and SLs in Corollary 4.15 below.

3.2 Coalgebraic Bisimilarity with Effects

By introducing effects through M, our coalgebraic definition of behavioural equivalence allows us to characterise a wide class of behaviours in a uniform manner, and we illustrate with the examples anticipated in Section 2.2.

3.2.1 Interactive I/O

We take $MX = \nu Y\,.\, X + (O \times Y) + Y^I$, where O is a given set of observable outputs, and I inputs the user can provide. The components of the disjoint union induce monadic $\mathsf{return} : X \to MX$ and algebraic operations $\mathsf{out} : O \times MX \to MX$ and $\mathsf{in} : (I \to MX) \to MX$ (c.f. [23]). In the context of cbx that exhibit I/O effects in this way, an operation like $set_L : X \to (MX)^A$ maps a state $x : X$ and an A-value $a : A$ to a value $m : MX$, where m describes some path in an (unbounded) tree of I/O actions, either terminating eventually and returning a new state in X, or diverging (depending on the user's input).

One may characterise pointed bisimulations on such cbx as follows. Intuitively, behaviourally equivalent states must "exhibit the same observable I/O activity" during updates set_L and set_R, and subsequently arrive at behaviourally equivalent states. To formalise this notion of I/O activity, we need an auxiliary definition (which derives from the greatest-fixpoint definition of M):

Definition 3.8. *With respect to an I/O monad M and a relation $R \subseteq X \times Y$, the I/O-equivalence relation $\sim_R \subseteq MX \times MY$ induced by R is the greatest equivalence relation such that $m \sim_R n$ whenever:*

- *$m = \mathsf{return}\ x$, $n = \mathsf{return}\ y$, and $(x, y) \in R$ for some x, y; or*
- *$m = \mathsf{out}\ (o, m')$ and $n = \mathsf{out}\ (o, n')$ for some $o : O$ and $m' \sim_R n'$; or*
- *$m = \mathsf{in}\ (\lambda i \to m(i))$ and $n = \mathsf{in}\ (\lambda i \to n(i))$, where $m(i) \sim_R n(i)$ for all $i : I$.*

One may now show that a pointed F^M_{AB}-bisimulation R on a pair of such cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ iff

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- for all $a : A$ and $b : B$, $(x.set_L\ a) \sim_R (y.set_L\ a)$ and $(x.set_R\ b) \sim_R (y.set_R\ b)$.

Such an equivalence guarantees that, following any sequence of updates in α or β, the user experiences exactly the same sequence of I/O actions; and when the sequence is complete, they observe the same values of A and B for either bx. Thus, pointed bisimulation asserts that α, β are indistinguishable from the user's point of view.

3.2.2 Failure

Here we take $MX = 1 + X$, and write None and $\mathsf{Just}\ x$ for the corresponding components of the coproduct. This induces a simple equivalence on pairs of bx, asserting that sequences of updates to either bx will succeed or fail in the same way. More formally, a pointed bisimulation R on a pair of coalgebraic bxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ iff for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- if $x.set_L\ a = \mathsf{None}$ then $y.set_L\ a = \mathsf{None}$ – and vice versa;
- if $x.set_L\ a = \mathsf{Just}\ x'$, then $y.set_L\ a = \mathsf{Just}\ y'$ for some y' with $(x', y') \in R$ – and vice versa;
- two analogous clauses with B and set_R substituted for A and set_L.

3.2.3 Non-determinism

Taking M to be the finitary powerset monad, the resulting behavioural equivalence on bx comes close to the standard notion of strong bisimulation on labelled transition systems – and as we will see, shares its excessive fine-grainedness. A pointed F^M_{AB}-bisimulation R on a pair of cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ iff for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$;
- for all $a : A$ and $x' \in x.set_L\ a$, there is some $y' \in y.set_L\ a$ with $(x', y') \in R$;
- for all $a : A$ and $y' \in y.set_L\ a$, there is some $x' \in x.set_L\ a$ with $(x', y') \in R$;
- three analogous clauses with B and get_R/set_R substituted for A and get_L/set_L.

In contrast with the case of user I/O, this equivalence may be too fine for comparing bx behaviours, as it exposes too much information about when non-determinism occurs. Here is a prototypical scenario: consider the effect of two successive L-updates. In one implementation, suppose an update $set_L\ a$ changes the system state from s to t, and a second update $set_L\ a'$ changes it to either u or u'. Each state-change is only observable to the user through the values of get_L and get_R; so suppose $u.get_R = u'.get_R = b$. (Note that $u.get_L = u'.get_L = a'$ by $(\mathsf{CGetSet}_L)$). This means u and u' cannot be distinguished by their *get* values.

In a different implementation, suppose $set_L\ a$ instead maps s to one of two states t' or t'' (both with the same values of get_R and get_L as state t above), and then $set_L\ a'$ maps these respectively to u and u' again. The states called s in both implementations, although indistinguishable to any user by observing their *get* values, are not bisimilar. In such situations, a coarser 'trace-based' notion of equivalence may be more appropriate [14].

4 Coalgebraic bx Composition

A cbx $\alpha : A \nrightarrow_X B$ describes how changes to a data source A are propagated across X to B (and vice versa). It is then natural to suppose, given another such $\beta : B \nrightarrow_Y C$, that we may propagate these changes to C (and vice versa), raising the question of whether there exists a composite cbx $\alpha \bullet \beta : A \nrightarrow_Z C$ for some Z. Here, we give a more general, coalgebraic definition of cbx composition than our previous account for mbx [1]; this lets us reason about behavioural equivalence of such compositions in Section 4.1.

First, we introduce some necessary technical details regarding weak pullback preserving (wpp) functors. Wpp functors are closed under composition, and we also exploit the following fact:

Lemma 4.1. *A wpp functor preserves monomorphisms, or "monos" (maps f which are 'right cancellable': $g.f = h.f$ implies $g = h$) [25, Lemma 4.4].*

Remark 4.2. *The following technical observation will also be useful for reasoning about F^M_{AB}-coalgebras. As M is wpp (assumption 2 of Section 2.1), so too is F^M_{AB}, using the fact that $A \times (-)$ and $(-)^A$ preserve pullbacks, and hence are wpp. Then by Lemma 4.1, F^M_{AB} also preserves monos.*

Finally, we will employ the following in proofs, where $k\ x\ a$ is a block of statements referring to x and a:

Lemma 4.3. $(\mathsf{CSetGet}_L)$ *and* $(\mathsf{CGetSet}_L)$ *are equivalent to the following 'continuation' versions:*

$$(\mathsf{CGetSet}_L)\ (\alpha) : \mathbf{do}\ \{\mathbf{let}\ a = x.get_L; x' \leftarrow x.set_L\ a; k\ x'\ a\} = \mathbf{do}\ \{\mathbf{let}\ a = x.get_L; k\ x\ a\}$$
$$(\mathsf{CSetGet}_L)\ (\alpha) : \mathbf{do}\ \{x' \leftarrow x.set_L\ a; k\ x'\ (x'.get_L)\} \qquad = \mathbf{do}\ \{x' \leftarrow x.set_L\ a; k\ x'\ a\}$$

Similarly, there are continuation versions of the coalgebra-morphism laws $(\mathsf{CGetP}_L)(h)$, $(\mathsf{CSetP}_L)(h)$, etc. in Remark 3.2, which we omit. We are now ready to define cbx composition; we do this in four stages.

(i) Defining a State-space for the Composition of α and β

The state-spaces of coalgebraic bx $\alpha : A \nleftarrow_X B, \beta : B \nleftarrow_Y C$ both contain information about B, in addition to A and C respectively. As indicated above, we define the state-space Z of the composite as consisting of those (x, y) pairs which are 'B-consistent', in that $x.get_R = y.get_L$. We must also identify an initial state in Z; the obvious choice is the pair $(\epsilon_\alpha, \epsilon_\beta)$ of initial states from α and β. To lie in Z, this pair itself must be B-consistent: $\epsilon_\alpha.get_R = \epsilon_\beta.get_L$. We may only compose cbx whose initial states are B-consistent in this way.

We now give the categorical formulation of these ideas, in terms of pullbacks:

Definition 4.4. *Given two pointed cbx $\alpha{:}A \nleftarrow_X B$ and $\beta{:}B \nleftarrow_Y C$, we define a* state-space *for their composition $\alpha \bullet \beta$ to be the pullback $P_{\alpha,\beta}$ in the below-left diagram. It is straightforward to show that this also makes the below-right diagram (also used in Step ((iii)) below) into a pullback, where $e_{\alpha,\beta}$ is defined to be $\langle p_\alpha, p_\beta \rangle$.*

$$\begin{array}{ccc} P_{\alpha,\beta} & \xrightarrow{p_\beta} & Y \\ \downarrow{\scriptstyle p_\alpha} & & \downarrow{\scriptstyle \beta.get_L} \\ X & \xrightarrow{\alpha.get_R} & B \end{array} \qquad \begin{array}{ccc} P_{\alpha,\beta} & \xrightarrow{e_{\alpha,\beta}=\langle p_\alpha, p_\beta \rangle} & X \times Y \\ \downarrow{\scriptstyle p_\alpha} & & \downarrow{\scriptstyle \alpha.get_R \times \beta.get_L} \\ X & \xrightarrow{\langle \alpha.get_R, \alpha.get_R \rangle} & B \times B \end{array}$$

For instance, in the category Set, these definitions may be interpreted as follows:

$$P_{\alpha,\beta} = \{\, (x, y) \mid x.get_R = y.get_L \,\} = \{\, (x, y) \mid (x.get_R, y.get_L) = (x.get_R, x.get_R) \,\}$$

and $\epsilon_{\alpha\bullet\beta}$ is the pair of initial states $(\epsilon_\alpha, \epsilon_\beta)$, assuming $\epsilon_\alpha.get_R = \epsilon_\beta.get_L$.

Remark 4.5. *Towards proving properties of the composition $\alpha \bullet \beta$ in Section 3, we note that $e_{\alpha,\beta}$ is also the* equalizer *of the parallel pair of arrows $\alpha.get_R \circ \pi_1, \beta.get_L \circ \pi_2 : X \times Y \to B$. Hence, a fortiori $e_{\alpha,\beta}$ is monic (i.e. left-cancellable), and thus by Lemma 4.1, so too is its image under the wpp functors M and F^M_{AB}.*

This allows us to pick out an initial state for Z, by noting that the arrow $(\epsilon_\alpha, \epsilon_\beta) : I \to X \times Y$ equalizes the parallel pair of morphisms in Remark 4.5; universality then gives the required arrow $\epsilon_{\alpha\bullet\beta} : I \to Z$.

(ii) Defining Pair-based Composition $\alpha \diamond \beta$

Definition 4.6. *$(X \times Y, \alpha \diamond \beta)$ is an F^M_{AC}-coalgebra with L-operations (similarly for R):*

$$(x, y).get_L = x.get_L \qquad (x, y).set_L\ a = \mathbf{do}\, \{\, x' \leftarrow x.set_L\ a; y' \leftarrow y.set_L\ (x'.get_R); \mathit{return}\ (x', y') \,\}$$

(iii) Inducing the Coalgebra $\alpha \bullet \beta$ on the Pullback

We now prove that the *set* operations of $\alpha \diamond \beta$ produce B-consistent pairs – even if the input pairs (x, y) were not B-consistent (because the *set* operations involve retrieving a B-value from one bx, and setting the same value in the other). Note that this implies $\alpha \diamond \beta$ will generally fail to be a coalgebraic bx, as it will not satisfy the coalgebraic bx law ($\mathsf{CGetSet}$): getting and then setting A or C in a B-inconsistent state will result in a different, B-consistent state, in contrast to the law's requirement that this should not change the state.

Lemma 4.7. *The following equation $(\dagger_L)$ holds at L for the set_L operation of Definition 4.6, and a corresponding property $(\dagger_R)$ for set_R. (The last two occurrences of $x'.get_R$ may equivalently be replaced with $y'.get_L$.)*

$$\begin{array}{rl} & \mathbf{do}\, \{\, (x', y') \leftarrow (x, y).set_L\ a; \mathit{return}\ (x'.get_R, y'.get_L) \,\} \\ = & \mathbf{do}\, \{\, (x', y') \leftarrow (x, y).set_L\ a; \mathit{return}\ (x'.get_R, x'.get_R) \,\} \end{array} \qquad (\dagger_L)$$

Proof. We prove $(\dagger_L)$; the argument for $(\dagger_R)$ is symmetric.

$$\begin{array}{ll} & \mathbf{do}\, \{\, (x', y') \leftarrow (x, y).set_L\ a; \mathit{return}\ (x'.get_R, y'.get_L) \,\} \\ = & [\![\ \text{definition of } (x, y).set_L\ ]\!] \\ & \mathbf{do}\, \{\, x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ (x'.get_R, y'.get_L) \,\} \\ = & [\![\ (\mathsf{CSetGet})\ (\beta), \text{ where } k\ y'\ b \text{ is } \mathit{return}\ (x'.get_R, b) \text{ (N.B. } k \text{ doesn't use } y')\ ]\!] \\ & \mathbf{do}\, \{\, x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ (x'.get_R, b) \,\} \\ = & [\![\ \text{undo inlining of } \mathbf{let}\ b = x'.get_R\ ]\!] \\ & \mathbf{do}\, \{\, x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ (x'.get_R, x'.get_R) \,\} \\ = & [\![\ \text{definition of } (x, y).set_L\ ]\!] \\ & \mathbf{do}\, \{\, (x', y') \leftarrow (x, y).set_L\ a; \mathit{return}\ (x'.get_R, x'.get_R) \,\} \end{array}$$

□

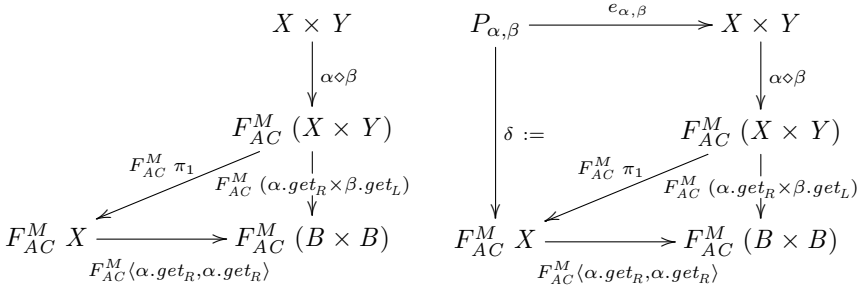
Remark 4.8. *In general, this is a stronger constraint than the corresponding equation*

$$\mathbf{do}\,\{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R)\} = \mathbf{do}\,\{(x',y') \leftarrow (x,y).set_L\ a; return\ (y'.get_L)\} \qquad (\dagger_L^*)$$

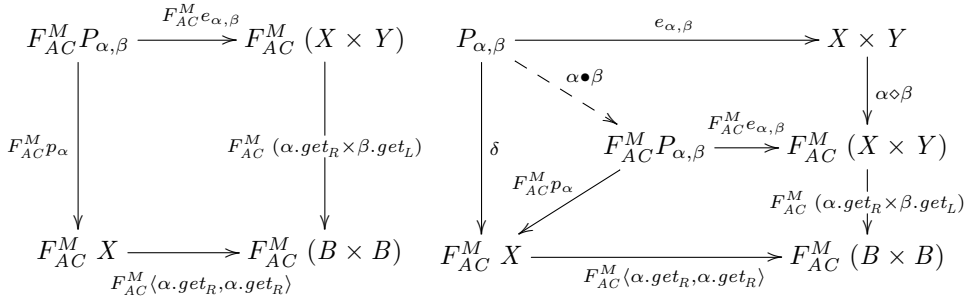
although it is equivalent if the monad M preserves jointly monic pairs (Definition 3.3). To illustrate the difference, suppose $B = \{0,1\}$ and consider a non-deterministic setting, where M is the (finitary) powerset monad on Set *(and indeed, it does not preserve jointly monic pairs). In state (x,y), suppose that $(set_L\ a)$ can land in either of two new states (x_1,y_1) or (x_2,y_2), where $x_2.get_R = y_1.get_L = 0$ and $x_1.get_R = y_2.get_L = 1$. Then $(\dagger_L^*)$ holds at (x,y) as both sides give $\{0,1\}$, but $(\dagger_L)$ does not, because the left side gives $\{(0,1),(1,0)\}$ and the right gives $\{(0,0),(1,1)\}$. We require the stronger version $(\dagger_L)$ to correctly define composition below.*

Our goal is to show that the properties $(\dagger_L)$ and $(\dagger_R)$ together are sufficient to ensure that the operations of $\alpha \diamond \beta : X \times Y \to F_{AC}^M\ (X \times Y)$, restricted to the B-consistent pairs $P_{\alpha,\beta}$, induce well-defined operations $P_{\alpha,\beta} \to F_{AC}^M P_{\alpha,\beta}$ on the pullback.

To do this, it is convenient to cast the properties $(\dagger_L)$, $(\dagger_R)$ in diagrammatic form, as shown in the left-hand diagram below. (It also incorporates two vacuous assertions, $(x,y).get_L = (x,y).get_L$ and similarly at R, which we may safely ignore.) Then, we precompose this diagram with the equalizer $e_{\alpha,\beta}$ as shown below-right, defining δ to be the resulting arrow $P_{\alpha,\beta} \to F_{AB}^M\ X$ given by the composition $F_{AB}^M\ \pi_1 \circ (\alpha \diamond \beta) \circ e_{\alpha,\beta}$.



Under the assumption that M is wpp, so is F_{AC}^M. Hence, the image under F_{AC}^M of the 'alternative' pullback characterisation of $P_{\alpha,\beta}$ (the right-hand diagram in Definition 4.4) is a weak pullback; it is shown below-left. Now the above-right diagram contains a cone over the same span of arrows; hence (by definition) we obtain a mediating morphism $P_{\alpha,\beta} \to F_{AC}^M\ (P_{\alpha,\beta})$ (not *a priori* unique) as shown below-right. We take this to be the coalgebra structure $\alpha \bullet \beta$ of the composite bx.



Although this does not explicitly define the operations of the composition $\alpha \bullet \beta$, it does relate them to those of $\alpha \diamond \beta$ via the monic arrow $F_{AC}^M\, e_{\alpha,\beta}$ (Remark 4.5) – allowing us to reason in terms of B-consistent pairs $(x,y) : X \times Y$, appealing to left-cancellability of monos. Moreover, in spite of only *weak* pullback preservation of F_{AB}^M, the coalgebra structure $\alpha \bullet \beta$ is canonical: there can be at most one coalgebra structure $\alpha \bullet \beta$ such that $e_{\alpha,\beta}$ is a coalgebra morphism from $\alpha \bullet \beta$ to $\alpha \diamond \beta$. This is a simple corollary of Lemma 4.11 below.

(iv) Proving the Composition is a Coalgebraic bx

Proposition 4.9. (CGetSet) $(\alpha \bullet \beta)$ *and* (CSetGet) $(\alpha \bullet \beta)$ *hold at L and R.*

Proof. We focus on the L case (the R case is symmetric). As described in (iii) above, we prove the laws post-composed with the monos $M e_{\alpha,\beta}$ and $M\ (e_{\alpha,\beta} \times id)$ respectively; left-cancellation completes the proof. (The law $(\mathsf{CSetP}_L)(e_{\alpha,\beta})$ is given after Definition 3.1 below.) Here is $(\mathsf{CGetSet}_L)$ $(\alpha \bullet \beta)$ postcomposed with $M e_{\alpha,\beta}$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\mathbf{let}\ a = z.get_L; z' \leftarrow z.set_L\ a; \mathit{return}\ (e_{\alpha,\beta}\ (z'))\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ a = z.get_L; \mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); (x',y') \leftarrow (x,y).set_L\ a; \mathit{return}\ (x',y')\} \\
= & \quad [\![\ \text{swapping } \mathbf{let}\text{s, and using } (\mathsf{CGetP}_A)(e_{\alpha,\beta})\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ a = (x,y).get_L; (x',y') \leftarrow (x,y).set_L\ a; \mathit{return}\ (x',y')\} \\
= & \quad [\![\ \text{definitions of } (x,y).get_L \text{ and } (x,y).set_L\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ a = x.get_L; x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ (x',y')\} \\
= & \quad [\![\ (\mathsf{CGetSet}_A) \text{ for } \alpha\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ b = x.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ (x,y')\} \\
= & \quad [\![\ (x,y) = e_{\alpha,\beta}\ (z) \text{ implies } x.get_L = y.get_R \text{ by definition of } e_{\alpha,\beta}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ b = y.get_L; y' \leftarrow y.set_L\ b; \mathit{return}\ (x,y')\} \\
= & \quad [\![\ (\mathsf{CGetSet}_L) \text{ for } \beta\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); \mathit{return}\ (x,y)\} \\
= & \quad [\![\ \text{inline } \mathbf{let}\text{; } \mathbf{do}\text{-laws}\ ]\!] \\
& \mathit{return}\ e_{\alpha,\beta}\ (z)
\end{array}
$$

$(\mathsf{CSetGet}_L)$ postcomposed with $M\ (e_{\alpha,\beta} \times id)$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{z' \leftarrow z.set_L\ (a); \mathit{return}\ (e_{\alpha,\beta}\ (z'), z'.get_L)\} \\
= & \quad [\![\ \text{inlining } \mathbf{let}\text{; definition of } z'.get_L\ ]\!] \\
& \mathbf{do}\,\{z' \leftarrow z.set_L\ (a); \mathbf{let}\ (x',y') = e_{\alpha,\beta}\ (z'); \mathit{return}\ ((x',y'), x'.get_L)\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); (x',y') \leftarrow (x,y).set_L\ a; \mathit{return}\ ((x',y'), x'.get_L)\} \\
= & \quad [\![\ \text{definition of } (x,y).set_L\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ ((x',y'), x'.get_L)\} \\
= & \quad [\![\ (\mathsf{CSetGet}_L) \text{ for } \alpha\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; \mathit{return}\ ((x',y'), a)\} \\
= & \quad [\![\ \text{definition of } (x,y).set_L\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (z); (x',y') \leftarrow (x,y).set_L\ a; \mathit{return}\ ((x',y'), a)\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\text{; inline } \mathbf{let}\ ]\!] \\
& \mathbf{do}\,\{z' \leftarrow z.set_L\ a; \mathit{return}\ (e_{\alpha,\beta}\ (z'), a)\}
\end{array}
$$

□

4.1 Well-behavedness of cbx Composition

Having defined a notion of composition for cbx, we must check that it has the properties one would expect, in particular that it is associative and has left and right identities. However, as noted in the Introduction, we cannot expect these properties to hold 'on the nose', but rather only up to some notion of behavioural equivalence. We will now prove that cbx composition is well-behaved up to the equivalence $\equiv$ introduced in Section 3 (Definition 3.5). Recall also the identity cbx $\iota\ (a) : A \nrightarrow A$ introduced in Example 2.7.

Theorem 4.10. *Coalgebraic bx composition satisfies the following properties (where $\alpha, \alpha' : A \nrightarrow B$, $\beta, \beta' : B \nrightarrow C$, and $\gamma, \gamma' : C \nrightarrow D$, and all compositions are assumed well-defined):*

identities: $\iota(\epsilon_\alpha.get_L) \bullet \alpha \equiv \alpha$ *and* $\alpha \bullet \iota(\epsilon_\alpha.get_R) \equiv \alpha$

congruence: *if* $\alpha \equiv \alpha'$ *and* $\beta \equiv \beta'$ *then* $\alpha \bullet \beta \equiv \alpha' \bullet \beta'$

associativity: $(\alpha \bullet \beta) \bullet \gamma \equiv \alpha \bullet (\beta \bullet \gamma)$

To prove this, we typically need to exhibit a coalgebra morphism from some coalgebra α to a composition $\beta = \psi \bullet \varphi$. As the latter is defined implicitly by the equalizer $e_{\psi,\varphi}$ – which is a *monic* coalgebra morphism from β to $\gamma = \psi \diamond \varphi$ – it is usually easier to reason by instead exhibiting a coalgebra morphism from α into $\gamma = \psi \diamond \varphi$, and then appealing to the following simple lemma:

Lemma 4.11. *Let F be wpp, and $a : \alpha \to \gamma \leftarrow \beta : b$ a cospan of pointed F-coalgebra morphisms with b monic. Then any $m : \alpha \to \beta$ with $b \circ m = a$ is also a pointed F-coalgebra morphism. If a is monic, than so is m; and for any q with (a, q) jointly monic, so is (m, q).*

Proof. One may show that $Fb \circ (\beta \circ m) = Fb \circ (Fm \circ \alpha)$ by a routine diagram-chase. Then using the fact that F preserves the mono b, we may left-cancel Fb on both sides. Moreover, if $m \circ f = m \circ f'$, then post-composing with b (and applying $b \circ m = a$) we obtain $a \circ f = a \circ f'$; the result then follows. □

Remark 4.12. *In the following proof, we will often apply Lemma 4.11 in the situation where b is given by an equalizer (such as $e_{\psi,\varphi}$, after Definition 3.5) which is also a coalgebra morphism, and where the coalgebra morphism a also equalizes the relevant parallel pairs. Then we obtain the arrow m by universality, and the Lemma ensures it is also a coalgebra morphism.*

We also use this simple fact in the first part of the proof (identities for composition):

Remark 4.13. *A* monic *pointed coalgebra morphism h from α to α' trivially yields a bisimulation by taking $(\zeta, p, q) = (\alpha, id, h)$, and hence $\alpha \equiv \alpha'$; but not all bisimulations arise in such a way.*

We are now ready for the proof of Theorem 4.10.

Proof. The general strategy is to prove that two compositions $\gamma = \delta \bullet \vartheta$ and $\gamma' = \delta' \bullet \vartheta'$ are $\equiv$-equivalent, by providing a jointly monic pair of pointed coalgebra morphisms p, q from some ζ into $\delta \diamond \vartheta$ and $\delta' \diamond \vartheta'$ respectively, which equalize the relevant parallel pairs. Lemma 4.11 and Remark 4.12 then imply the existence of a jointly monic pair of pointed coalgebra morphisms m, m' into $\delta \bullet \vartheta$ and $\delta' \bullet \vartheta'$, giving the required bisimulation. We indicate the key steps (i), (ii), etc. in each proof below.

identities: We sketch the strategy for showing $\iota(\epsilon_\alpha.get_L) \bullet \alpha \equiv \alpha$, as the other identity is symmetric. We exhibit the equivalence by taking α itself to be the coalgebra defining a bisimulation between α and $\iota(\epsilon_\alpha.get_L) \bullet \alpha$. To do this, one shows that (i) $h = \langle \alpha.get_L, id \rangle : X \to A \times X$ is a pointed coalgebra morphism from α to the composition $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ defined on pairs, and (ii) h also equalizes the parallel pair $\iota(\epsilon_\alpha.get_L).get_R \circ \pi_1$ and $\alpha.get_L \circ \pi_2$ (characterising the equalizer and coalgebra morphism from $\iota(\epsilon_\alpha.get_L) \bullet \alpha$ to $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ – see Remark 4.12). Moreover, h is easily shown to be monic. Hence by Lemma 4.11, h induces a pointed coalgebra morphism from α to $\iota(\epsilon_\alpha.get_L) \bullet \alpha$ which is also monic (in $\mathbb{C}$), and hence by Remark 4.13 we obtain the required bisimulation.
As for pointedness, the initial state of $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ is $(\epsilon_\alpha.get_L, \epsilon_\alpha)$, and this is indeed $h\ (\epsilon_\alpha)$ as required.

congruence: We show how to prove that right-composition $(- \bullet \beta)$ is a congruence: i.e. that $\alpha \equiv \alpha'$ implies $(\alpha \bullet \beta) \equiv (\alpha' \bullet \beta)$. By symmetry, the same argument will show left-composition $(\alpha' \bullet -)$ is a congruence. Then one may use the standard fact that 'bisimulations compose (for wpp functors)': given pointed bisimulations between γ and δ, and δ and ε, one may obtain a pointed bisimulation between γ and ε – provided the behaviour functor F^M_{AC} is wpp, which follows from our assumption that M is wpp. This allows us to deduce that composition is a congruence in both arguments simultaneously, as required.
So, suppose given a pointed bisimulation between α and α': an F^M_{AB}-coalgebra (R, r) with a jointly monic pair p, p' of pointed coalgebra morphisms from r to α, α' respectively. One exhibits a bisimulation between $\alpha \bullet \beta$ and $\alpha' \bullet \beta$ as follows, by first constructing a suitable coalgebra (S, s), together with a jointly monic pair (q, q') of coalgebra morphisms from s to the compositions $\alpha \bullet \beta$, $\alpha' \bullet \beta$. To construct s, let ζ be the equalizer of the following parallel pair – or equivalently, the pullback of $r.get_R$ and $\beta.get_L$.

$$S \dashrightarrow R \times Y \underset{\beta.get_L \circ \pi_2}{\overset{r.get_R \circ \pi_1}{\rightrightarrows}} B$$

One may then follow the steps (i)-(iii) in Section 4, where ζ and r play the role of the equalizer $e_{\alpha,\beta}$ and β respectively, to construct s, such that ζ is a coalgebra morphism from s to $r \diamond \beta$. Even though r is not a coalgebraic bx, it satisfies the following weaker form of $(\mathsf{CSetGet}_L)$ (and its R-version), sufficient for Lemma 4.7 to hold. (Here, we take $j : R$ and $a : A$; there is a continuation version as in Lemma 4.3.)

$$(\mathsf{CSetGet}^-_L)(r) : \mathbf{do}\ \{j' \leftarrow j.set_L\ a; return\ j'.get_L\} = \{j' \leftarrow j.set_L\ a; return\ a\}$$

To construct $q : S \to P_{\alpha,\beta}$, it is enough to show (i) the composition of the upper and right edges in the following diagram equalizes the given parallel pair:

$$\begin{array}{ccccc} S & \xrightarrow{\quad\zeta\quad} & R \times Y & & \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle p \times id} & & \\ P_{\alpha,\beta} & \xrightarrow{\quad e_{\alpha,\beta}\quad} & X \times Y & \underset{\beta.get_L \circ \pi_2}{\overset{\alpha.get_R \circ \pi_1}{\rightrightarrows}} & B \end{array}$$

By also showing that (ii) $p \times id$ is in fact a coalgebra morphism, we can then appeal to Lemma 4.11 to show that q itself defines a coalgebra morphism from s into the composition $\alpha \bullet \beta$. One obtains the coalgebra morphism q' from s to $\alpha' \bullet \beta$ in an analogous way. Finally, from p, p' being jointly monic, and the fact that $e_{\alpha,\beta}$ is an equalizer, we obtain a proof that q, q' are jointly monic.

associativity: We follow the same strategy as we did for proving the identity laws: we may prove this law by providing a pointed coalgebra morphism p from the LHS composition to the RHS, which is moreover monic. We will do this in two stages: first, we show how to obtain an arrow $p_0 : P_{(\alpha\bullet\beta),\gamma} \to X \times P_{\beta,\gamma}$ making the square in the following diagram commute; then, by applying Lemma 4.11 and Remark 4.12, we will show it is a pointed coalgebra morphism from $(\alpha \bullet \beta) \bullet \gamma$ to $\alpha \diamond (\beta \bullet \gamma)$, which is also monic.

$$\begin{array}{ccccc} P_{(\alpha\bullet\beta),\gamma} & \xrightarrow{e_{(\alpha\bullet\beta),\gamma}} & P_{\alpha,\beta} \times Z & & \\ \downarrow{\scriptstyle p_0} & & \downarrow{\scriptstyle f} & & \\ X \times P_{\beta,\gamma} & \xrightarrow{id\times e_{\beta,\gamma}} & X \times (Y \times Z) & \underset{id\times(\gamma.get_L\circ\pi_2)}{\overset{id\times(\beta.get_R\circ\pi_1)}{\rightrightarrows}} & X \times C \end{array} \tag{1}$$

In this diagram, the arrow f is defined by $f\ (u, z) = \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); return\ (x, (y, z))\}$, but it may also be expressed as $f = assoc \circ (e_{\alpha,\beta} \times id)$, where we write *assoc* for the associativity isomorphism on products; the isomorphism, and the pairing with *id*, make it apparent that f is monic. Note that the functor $X \times (-)$, being a right adjoint, preserves equalizers, and hence $(id \times e_{\beta,\gamma})$ is the equalizer of the given parallel pair (and moreover monic).

Following the proof strategy outlined above, to obtain p_0 one must show that: (i) $f \circ e_{(\alpha\bullet\beta),\gamma}$ equalizes the parallel pair in the above diagram, ensuring existence of an arrow p_0 making the square commute; (ii) the equalizer $id \times e_{\beta,\gamma}$ is a pointed coalgebra morphism from $\alpha \diamond (\beta \bullet \gamma)$ to $\alpha \diamond (\beta \diamond \gamma)$; and (iii) f is a pointed coalgebra morphism from $(\alpha \bullet \beta) \diamond \gamma$ to $\alpha \diamond (\beta \diamond \gamma)$. The facts (ii), (iii) allow us to apply Lemma 4.11 and Remark 4.12, to deduce that p_0 is a pointed coalgebra morphism, and moreover monic (using the fact that f and $e_{(\alpha\bullet\beta),\gamma}$ are, and hence their composition too).

The final stage of constructing the required arrow $p : P_{(\alpha\bullet\beta),\gamma} \to P_{\alpha,(\beta\bullet\gamma)}$ is to show that: (iv) p_0 equalizes the parallel pair defining $P_{\alpha,(\beta\bullet\gamma)}$ as shown below. Thus we obtain p as the mediating morphism into the equalizer $P_{\alpha,(\beta\bullet\gamma)}$; by Remark 4.12 it is a coalgebra morphism, and it is monic because p_0 is monic.

$$\begin{array}{ccccc} P_{(\alpha\bullet\beta),\gamma} & & & & \\ \vdots\ {\scriptstyle p} & \searrow^{p_0} & & & \\ P_{\alpha,(\beta\bullet\gamma)} & \xrightarrow{e_{\alpha,(\beta\bullet\gamma)}} & X \times P_{\beta,\gamma} & \underset{(\beta\bullet\gamma).get_L\circ\pi_2}{\overset{\alpha.get_R\circ\pi_1}{\rightrightarrows}} & B \end{array}$$

□

This well-behavedness of composition allows us to define a category of cbx:

Corollary 4.14. *There is a category* $\mathsf{Cbx}_\bullet$ *of pointed cbx, whose objects are* pointed sets $(A, a : A)$ *– sets with distinguished, "initial" values* a *– and whose arrows* $(A, a) \to (B, b)$ *are* $\equiv$*-equivalence classes* $[\alpha]$ *of coalgebraic bx* $\alpha : A \nrightarrow B$ *satisfying* $\epsilon_\alpha.get_L = a$ *and* $\epsilon_\alpha.get_R = b$.

We now describe how this category is related (by taking $\mathbb{C} = \mathsf{Set}$ and $M = Id$) to the category of symmetric lenses defined in [16]. The point of departure is that cbx encapsulate *additional data*, namely initial values $\epsilon_\alpha.get_L, \epsilon_\alpha.get_R$ for A and B. The difference may be reconciled if one is prepared to extend SLs with such data (and consider distinct initial-values to give distinct SLs, *cf.* the comments beginning Section 2.2):

Corollary 4.15. *Taking* $\mathbb{C} = \mathsf{Set}$ *and* $M = Id$, $\mathsf{Cbx}_\bullet$ *is isomorphic to a subcategory of* SL*, the category of SL-equivalence-classes of symmetric lenses; and there is an isomorphism of categories* $\mathsf{Cbx}_\bullet \cong \mathsf{SL}_\bullet$ *where* $\mathsf{SL}_\bullet$ *is the category of (SL-equivalence-classes) of SLs additionally equipped with initial left- and right-values.*

5 Relating Coalgebraic and Monadic bx

Here, we consider the relationship between our coalgebraic notion of bx and our previous monadic account [1]. The latter characterised bx in terms of monadic *get* and *set* operations, where the monad was not assumed *a priori* to refer to any notion of state – such as the monad $StateT\ X\ M$, abbreviated to T_X^M throughout this paper. The monad was only restricted to this stateful form for defining bx composition; such monadic bx were

called *StateTBX*. By contrast, our coalgebraic bx are explicitly stateful from the outset. Therefore, to compare cbx and mbx in a meaningful way, throughout this Section we restrict mbx to *StateTBX*, i.e. with respect to the monad T_X^M (as in most of our leading examples of monadic bx). Moreover, as was necessary for defining composition for such mbx, we also assume them to be *transparent*: the *get* operations neither change the state nor introduce M-effects, i.e. $get_L = \lambda x.return\ (f\ x, x)$ for some $f :: X \to A$ (likewise get_R).

Finally, we also assume monadic bx $A \Longleftrightarrow B$ have explicit initial states, rather than the more intricate process of initialising by supplying an initial A- or B-value as in [1].

5.1 Translating a Coalgebraic bx into a Monadic bx

Given a cbx $\alpha : X \to F_{AB}^M\ X$, we can define its *realisation*, or "monadic interpretation", $[\![\alpha]\!]$ as a transparent *StateTBX* with the following operations. (Following conventional Haskell notation, we overload the symbol () to mean the unit type, as well as its unique value.)

$$
\begin{array}{ll}
[\![\alpha]\!].get_L : T_X^M\ A & \stackrel{\text{def}}{=} \mathbf{do}\ \{x \leftarrow get; return\ (x.get_L)\} \\
[\![\alpha]\!].get_R : T_X^M\ B & \stackrel{\text{def}}{=} \mathbf{do}\ \{x \leftarrow get; return\ (x.get_R)\} \\
[\![\alpha]\!].set_L : A \to T_X^M\ () & \stackrel{\text{def}}{=} \lambda a \to \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'\} \\
[\![\alpha]\!].set_R : B \to T_X^M\ () & \stackrel{\text{def}}{=} \lambda b \to \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_R\ b); set\ x'\}
\end{array}
$$

Here, the standard polymorphic functions associated with T_X^M, namely $get : \forall \alpha\,.\, T_\alpha^M\ \alpha$, $set : \forall \alpha\,.\, \alpha \to T_\alpha^M\ ()$, and the monad morphism $lift : \forall \alpha\,.\, M\alpha \to T_X^M\ \alpha$ (the curried form of the strength of M) are given by:

$$
\begin{array}{ll}
get = \lambda a \to return\ (a, a) & set = \lambda a'\ a \to return\ ((), a') \\
lift = \lambda ma\ x \to \mathbf{do}\ \{a \leftarrow ma; return\ (a, x)\}
\end{array}
$$

Proposition 5.1. *$[\![\alpha]\!]$ indeed defines a transparent StateTBX over T_X^M.*

Lemma 5.2. *The translation $[\![\cdot]\!]$ – from cbx for monad M with carrier X, to transparent StateTBX with an initial state – is surjective; it is also injective, using our initial assumption that return is monic.*

This fully identifies the subset of monadic bx which correspond to coalgebraic bx. Note that the translation $[\![\cdot]\!]$ is defined on an *individual* coalgebraic bx, not an equivalence class; we will say a little more about the respective categories at the end of the next section.

5.2 Composing Stateful Monadic bxs

We will review the method given in [1] for composing *StateTBX*, using monad morphisms induced by lenses on the state-spaces. We show that this essentially simplifies to step (ii) of our definition in Section 4 above; thus, our definition may be seen as a more categorical treatment of the set-based monadic bx definition.

Definition 5.3. *An* asymmetric lens *from* source A *to* view B, *written* $l : A \Rightarrow B$, *is given by a pair of maps* $l = (v, u)$, *where* $v : A \to B$ *(the* view *or* get *mapping) and* $u : A \times B \to A$ *(the* update *or* put *mapping). It is* well-behaved *if it satisfies the first two laws* (VU), (UV) *below, and* very well-behaved *if it also satisfies* (UU).

$$
\text{(VU)}\quad u\ (a, (v\ a)) = a \qquad \text{(UV)}\ v\ (u\ (a, b')) = b' \qquad \text{(UU)}\ u\ (u\ (a, b'), b'') = u\ (a, b'')
$$

Lenses have a very well-developed literature [6, 8, 16, 18, among others], which we do not attempt to recap here. For more discussion of lenses, see Section 2.5 of [1].

We will apply a mild adaptation of a result in Shkaravska's early work [28].

Proposition 5.4. *Let $l = (v, u) : Z \Rightarrow X$ be a lens, and define $\vartheta\ l$ to be the following natural transformation:*

$$
\begin{array}{l}
\vartheta\ l :: \forall \alpha\,.\,(Z \Rightarrow X) \to T_X^M\ \alpha \to T_Z^M\ \alpha \\
\vartheta\ ma \stackrel{\text{def}}{=} \mathbf{do}\ \{z \leftarrow get; (a', x') \leftarrow lift\ (ma\ (v\ z)); z' \leftarrow lift\ (u\ (z, x')); set\ z'; return\ a'\}
\end{array}
$$

If l is very well-behaved, then $\vartheta\ l$ is a monad morphism.

We apply Prop. 5.4 to the following two lenses, allowing views and updates of the projections from $X \times Y$:

$$\begin{array}{ll} l_1 = (v_1, u_1) : (X \times Y) \Rightarrow X & l_2 = (v_2, u_2) : (X \times Y) \Rightarrow Y \\ \quad v_1\ (x, y) \quad\ = x & \quad v_2\ (x, y) \quad\ = y \\ \quad u_1\ ((x, y), x') = (x', y) & \quad u_2\ ((x, y), y') = (x, y') \end{array}$$

This gives us a cospan of monad morphisms $\mathit{left} = \vartheta\ (l_1) : T^M_X \dot{\to} T^M_{(X \times Y)} \dot{\leftarrow} T^M_Y : \vartheta\ (l_2) = \mathit{right}$, allowing us to embed computations involving state-space X or Y into computations on the combined state-space $X \times Y$.

Now suppose we are given two $\mathit{StateTBX}$ $t_1 : A \Longleftrightarrow_{T^M_X} B$, $t_2 : B \Longleftrightarrow_{T^M_Y} C$ with

$$\begin{array}{llll} t_1.\mathit{get}_L :: T^M_X\ A & t_1.\mathit{get}_R :: T^M_X\ B & t_2.\mathit{get}_L :: T^M_Y\ B & t_2.\mathit{get}_R :: T^M_Y\ C \\ t_1.\mathit{set}_L :: A \to T^M_X\ () & t_1.\mathit{set}_R :: B \to T^M_X\ () & t_2.\mathit{set}_L :: B \to T^M_Y\ () & t_2.\mathit{set}_R :: C \to T^M_Y\ () \end{array}$$

In [1], we used $\mathit{left}, \mathit{right}$ to define $t_1 \fatsemi t_2$, a composite $\mathit{StateTBX}$ with state-space $X \times Y$, as follows:

$$\begin{array}{l} (t_1 \fatsemi t_2).\mathit{get}_L \quad = \mathit{left}\ (t_1.\mathit{get}_L) \qquad\qquad (t_1 \fatsemi t_2).\mathit{get}_R = \mathit{right}\ (t_2.\mathit{get}_R) \\ (t_1 \fatsemi t_2).\mathit{set}_L\ a = \mathbf{do}\ \{\mathit{left}\ (t_1.\mathit{set}_L\ a); b \leftarrow \mathit{left}\ (t_1.\mathit{get}_R); \mathit{right}\ (t_2.\mathit{set}_L\ b)\} \\ (t_1 \fatsemi t_2).\mathit{set}_R\ c = \mathbf{do}\ \{\mathit{right}\ (t_2.\mathit{set}_R\ c); b \leftarrow \mathit{right}\ (t_2.\mathit{get}_L); \mathit{left}\ (t_1.\mathit{set}_R\ b)\} \end{array}$$

We then defined the subset, $X \bowtie Y$, of $X \times Y$ given by B-consistent pairs, and argued that $t_1 \fatsemi t_2$ preserved B-consistency – hence its state-space could be restricted from $X \times Y$ to $X \bowtie Y$. We will use the notation $t_1 \bullet t_2$ for the resulting composite $\mathit{StateTBX}$.

In the context of coalgebraic bx, we made this part of the argument categorical by defining $X \bowtie Y$ to be a pullback, and formalising the move from the pairwise definition $\alpha \diamond \beta$ to the full composition $\alpha \bullet \beta$ by step (iii) of Section 4. Given the 1-1 correspondence between transparent $\mathit{StateTBX}$ and cbx given by Lemma 5.2, this may be considered to be the formalisation of the monadic move from the composite $t_1 \fatsemi t_2$ on the product state-space to the composite $t_1 \bullet t_2$ on the pullback.

This allows us to state our second principal result, namely that the two notions of composition – coalgebraic bx (as in Definition 4.4) and $\mathit{StateTBX}$ – may be reconciled by showing the pairwise definitions coherent:

Theorem 5.5. *Coherence of composition: The definitions of StateTBX and coalgebraic bx composition on product state-spaces are coherent:* $[\![\alpha]\!] \fatsemi [\![\beta]\!] = [\![\alpha \diamond \beta]\!]$. *Hence, the full definitions (on B-consistent pairs) are also coherent:* $[\![\alpha]\!] \bullet [\![\beta]\!] = [\![\alpha \bullet \beta]\!]$.

Proof. The operations of the monadic bx $[\![\alpha]\!]$ at the beginning of Section 5.1, and the computation $\vartheta\ (v, u)\ ma$, may be re-written in **do** notation for the monad M rather than in $\mathit{StateT}\ X\ M$, as follows:

$$\begin{array}{l} [\![\alpha]\!].\mathit{get}_L : T^M_X\ A \qquad [\![\alpha]\!].\mathit{set}_L : A \to T^M_X\ () \qquad \vartheta\ (v, u)\ ma : \forall \alpha\,.\ T^M_Z\ \alpha \\ [\![\alpha]\!].\mathit{get}_L = \lambda x \to \mathit{return}\ (x.\mathit{get}_L) \qquad [\![\alpha]\!].\mathit{set}_L\ a = \lambda x \to \mathbf{do}\ \{x' \leftarrow x.\mathit{set}_L\ a; \mathit{return}\ ((), x')\} \\ \vartheta\ (v, u)\ ma = \lambda z \to \mathbf{do}\ \{(a', s') \leftarrow ma\ (v\ z); \mathbf{let}\ z' = u\ (z, x'); \mathit{return}\ (a', z')\} \end{array}$$

Applying ϑ to the lenses l_1 and l_2 gives the monad morphisms left and right as follows:

$$\begin{array}{l} \mathit{left} \quad = \vartheta\ (l_1) : \forall \alpha\,.\ T^M_X\ \alpha \to T^M_{(X \times Y)}\ \alpha \qquad \mathit{right} = \vartheta\ (l_2) : \forall \alpha\,.\ T^M_Y\ \alpha \to T^M_{(X \times Y)}\ \alpha \\ \mathit{left} \quad = \lambda mxa\ (x, y) \to \mathbf{do}\ \{(a', x') \leftarrow mxa\ x; \mathit{return}\ (a', (x', y))\} \\ \mathit{right} = \lambda mya\ (x, y) \to \mathbf{do}\ \{(a', y') \leftarrow mya\ y; \mathit{return}\ (a', (x, y'))\} \end{array}$$

This allows us to unpack the operations of the composite $[\![\alpha]\!] \fatsemi [\![\beta]\!]$ to show they are the same as $[\![\alpha \diamond \beta]\!]$. □

Again, this result concerns individual cbx, and not the $\equiv$-equivalence classes used to define $\mathsf{Cbx}_\bullet$. We now comment on how one may embed the latter into a category of transparent $\mathit{StateTBX}$. In [1] (Theorem 26), we defined an equivalence on $\mathit{StateTBX}$ ($\sim$, say) given by operation-respecting state-space isomorphisms, and showed that $\bullet$-composition is associative up to $\sim$. In line with Corollary 4.14, one obtains a category $\mathsf{Mbx}_\bullet$ of $\sim$-equivalence-classes of transparent $\mathit{StateTBX}$, and initial states as at the start of Section 5).

Translating this into our setting (by reversing $[\![\cdot]\!]$ from Lemma 5.2), one finds that two transparent $\mathit{StateTBX}$ are $\sim$-equivalent iff there is an *isomorphism* of pointed coalgebra morphisms between their carriers – which is generally finer than $\equiv$. Therefore, we cannot hope for an equivalence-respecting embedding from $\mathsf{Cbx}_\bullet$ to $\mathsf{Mbx}_\bullet$. However, we may restrict $\equiv$ to make such an embedding possible:

Corollary 5.6. *Let $\equiv_!$ be the equivalence relation on coalgebraic bx given by restricting $\equiv$ to bisimulations whose pointed coalgebra morphisms are isomorphisms; and let $\mathsf{Cbx}^!_\bullet$ be the category of equivalence-classes of cbx up to $\equiv_!$. Then there is an isomorphism $\mathsf{Cbx}^!_\bullet \cong \mathsf{Mbx}_\bullet$.*

6 Conclusions and Further Work

In our search for unifying foundations of the field of bidirectional transformations (bx), we have investigated a number of approaches, including monadic bx, building on those of (symmetric) lenses and relational bx.

We have given a coalgebraic account of bx in terms of intuitively simple building blocks: two data sources A and B; a state space X; operations on X to observe (get_L, get_R) and update (set_L, set_R) each data source; an ambient monad M of additional computational effects; and a collection of laws, entirely analogous to those in existing bx formalisms. Moreover, these structures, being defined categorically, may be interpreted in a wide variety of settings under very modest assumptions about the underlying category $\mathbb{C}$.

Initial states were handled by introducing pointed coalgebras and bisimulations. A more elegant approach would be to work in the category $I \,/\, \mathbb{C}$ of *pointed objects* from the outset – i.e. pairs $(A, a : I \to A)$ of an object of $\mathbb{C}$ and an initial value a. Coalgebra morphisms and bisimulations are then automatically pointed, and the assumption of initial B-consistency used in Section 4 ((i)) is enforced at a type level. However, lifting exponentials (and strong monads) from $\mathbb{C}$ into $I \,/\, \mathbb{C}$ is rather delicate; we leave the details for future work.

Our definition allows a conceptually more direct, if technically slightly subtle, treatment of composition – in which the state space, defined by a pullback, captures the idea of communication across a shared data source, via the idea of B-consistent pairs. Our proof techniques involved reasoning about composition by considering operations defined on such pairs. We defined an equivalence on cbx based on coalgebraic bisimulation, and showed that composition does indeed define a category, up to equivalence. The notion of bisimulation improves on existing, more ad-hoc definitions, such as that of symmetric lens equivalence, and we take this as further evidence for the suitability of our framework and definitions.

We described several concrete instantiations of the general definition of bisimulation, given by varying the effect monad M. Coarser equivalences may be suitable for modelling non-deterministic bx, and could be introduced via alternative methods such as [14]; but we do not have space to explore this further here.

It is also worth investigating the relationship between our coalgebraic formulation of bx, and the spans of lenses considered by Johnson and Rosebrugh [20]. A coalgebraic bx may be seen as a generalisation to a span of 'monadic lenses' or 'M-lenses', allowing effectful updates. However, in defining equivalence for spans of lenses, Johnson and Rosebrugh consider an additional condition, namely that the mediating arrow between two spans, witnessing equivalence, is a lens in which the get arrow is a split epi. It is unclear to us the relationship between this condition, and pointed coalgebraic bisimulation. We leave such investigations to future work.

Another area to explore is combinatory structure for building larger cbx out of smaller pieces, in the style of existing work in the lens community. Some examples were given in [1] – of which bx composition was by far the most challenging to formulate – and we expect the other combinators to apply routinely for cbx as well, along the same lines as [4]. One would expect coalgebraic bisimulation to be a congruence for these combinators; it would be interesting to investigate whether the work of Turi and Plotkin [33] can help here.

We set great store by the fact that our definitions are interpretable in categories other than Set; we hope thereby to incorporate richer, more intensional structures, such as delta lenses [7], edit lenses [17] and ordered updates [15] in our framework, and not merely the state-based extensional bx we have considered so far. We also hope to explore the connections between our work and existing coalgebraic notions of software components [4], and techniques for composing coalgebras [13].

Acknowledgements An extended, unpublished abstract [2] of some of the results here was presented at CMCS 2014; we thank the participants there for useful feedback. Our work is supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* [32] (EP/K020218/1, EP/K020919/1); we are grateful to our colleagues for their support, encouragement and feedback during the writing of the present paper.

References

[1] F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Notions of bidirectional computation and entangled state monads. In *MPC*, June 2015. To appear. Extended version available at `http://groups.inf.ed.ac.uk/bx/bx-effects-tr.pdf`.

[2] F. Abou-Saleh and J. McKinna. A coalgebraic approach to bidirectional transformations. In *CMCS*. ETAPS, 2014. Talk abstract.

[3] J. Adámek, S. Milius, L. S. Moss, and L. Sousa. Well-pointed coalgebras. *Logical Methods in Computer Science*, 9(3), 2013.

[4] L. S. Barbosa. Towards a calculus of state-based software components. *J. UCS*, 9(8):891–909, 2003.

[5] J. Cheney, J. McKinna, P. Stevens, J. Gibbons, and F. Abou-Saleh. Entangled state monads (extended abstract). In Terwilliger and Hidaka [31].

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[7] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *JOT*, 10:6: 1–25, 2011.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007.

[9] J. Gibbons and R. Hinze. Just **do** it: Simple monadic equational reasoning. In *ICFP*, pages 2–14, 2011.

[10] J. Gibbons and M. Johnson. Relating algebraic and coalgebraic descriptions of lenses. *ECEASST*, 49, 2012.

[11] H. P. Gumm. Functors for coalgebras. *Algebra Universalis*, 45(2-3):135–147, 2001.

[12] H. P. Gumm. Copower functors. *TCS*, 410(12):1129–1142, 2009.

[13] I. Hasuo. The microcosm principle and compositionality of GSOS-based component calculi. In *Algebra and Coalgebra in Computer Science*, pages 222–236. Springer, 2011.

[14] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *CoRR*, abs/0710.2505, 2007.

[15] S. J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Art. Int.*, 40:63–125, 2004.

[16] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *POPL*, pages 371–384. ACM, 2011.

[17] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *POPL*, pages 495–508. ACM, 2012.

[18] Z. Hu, A. Schurr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar 11031: Bidirectional transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011. DOI: `10.4230/DagRep.1.1.42`.

[19] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin EATCS*, 62:222–259, 1997.

[20] M. Johnson and R. Rosebrugh. Spans of lenses. In Terwilliger and Hidaka [31].

[21] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[22] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[23] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FOSSACS*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.

[24] J. Power and O. Shkaravska. From comodels to coalgebras: State and arrays. *ENTCS*, 106, 2004.

[25] J. Power and H. Watanabe. An axiomatics for categories of coalgebras. *ENTCS*, 11:158–175, 1998.

[26] J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[27] A. Schürr and F. Klar. 15 years of triple graph grammars. In *ICGT*, volume 5214 of *LNCS*, pages 411–425. Springer, 2008.

[28] O. Shkaravska. Side-effect monad, its equational theory and applications. Seminar slides available at: `http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf`, 2005.

[29] A. Sokolova. Probabilistic systems coalgebraically: A survey. *TCS*, 412(38):5095–5110, 2011.

[30] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *SoSyM*, 9(1):7–20, 2010.

[31] J. Terwilliger and S. Hidaka, editors. *BX Workshop*. `http://ceur-ws.org/Vol-1133/#bx`, 2014.

[32] TLCBX Project. A theory of least change for bidirectional transformations. `http://www.cs.ox.ac.uk/projects/tlcbx/`, `http://groups.inf.ed.ac.uk/bx/`, 2013–2016.

[33] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291. IEEE, Computer Society Press, 1997.

Distributing Commas, and the Monad of Anchored Spans

Michael Johnson
Departments of Mathematics and Computing, Macquarie University
Robert Rosebrugh
Department of Mathematics and Computer Science
Mount Allison University

Abstract

Spans are pairs of arrows with a common domain. Despite their symmetry, spans are frequently viewed as oriented transitions from one of the codomains to the other codomain. The transition along an oriented span might be thought of as transitioning backwards along the first arrow (sometimes called 'leftwards') and then, having reached the common domain, forwards along the second arrow (sometimes called 'rightwards'). Rightwards transitions and their compositions are well-understood. Similarly, leftwards transitions and their compositions can be studied in detail. And then, with a little hand-waving, a span is 'just' the composite of two well-understood transitions — the first leftwards, and the second rightwards.

In this paper we note that careful treatment of the sources, targets and compositions of leftwards transitions can be usefully captured as a monad L built using a comma category construction. Similarly the sources, targets and compositions of rightwards transitions form a monad R, also built using a comma category construction. Our main result is the development of a distributive law, in the sense of Beck [3] but only up to isomorphism, distributing L over R. Such a distributive law makes RL a monad, the monad of anchored spans, thus removing the hand-waving referred to above, and establishing a precise calculus for span-based transitions.

As an illustration of the applicability of this analysis we use the new monad RL to recast and strengthen a result in the study of databases and the use of lenses for view updates.

1 Introduction

1.1 Set-based bidirectional transformations

There is an important distinction among extant bidirectional transformations between those that are "set-based" and those that are "category-based" (the latter are sometimes also called "delta-based"). This paper analyses the origins of that distinction and lays the mathematical foundations for a calculus integrating both points of view along with an often implicit point of view in which deltas sometimes have a "preferred" direction.

So, what are these set-based and category-based transformations, and how did the distinction arise?

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

To quote from the Call for Papers: "Bidirectional transformations (Bx) are a mechanism for maintaining the consistency of at least two related sources of information. Such sources can be relational databases, software models and code, or any other document following standard or ad-hoc formats."

A fundamental question in bidirectional transformations is: What are the *state spaces* of the sources among which consistency needs to be maintained? After all, consistency only needs to be worked on when one source changes state, so a careful analysis of permissible state changes is important.

This analysis will be important whatever the nature of the sources, but for ease of explication we will focus at first on an example. Let us consider relational databases. Recall that state spaces of systems are most often represented by graphs whose nodes are states and whose arrows represent transitions among the states.

Consider the state space of a relational database. Nodes in the state space, states, are just snapshots of the database at a moment in time — all of the data, stored in the database, in its structured form, at that moment.

Typically, with a database, one can transition (update the database) from any snapshot to any other. So, the state space could be all possible snapshots with an arrow between every pair of snapshots. (This kind of state space is sometimes called a "chaotic" or "co-discrete" category. A *co-discrete category* is a category with exactly one arrow between each ordered pair of objects.)

This kind of state-space, a co-discrete state-space, is a perfectly reasonable analysis of database updating. It comes naturally from focusing on the states, and from noticing that there is always an update that will lead from any state S to any other state S'. After all, when one can transition between any two states, why keep track of arrows that tell you that you can do that? All that one needs to know is what the new state S' is. And, no matter what the current state S might be, the database can transition to that new state S'.

So, it is natural to do away with the arrows, consider simply the set of states, and plan one's Bx assuming that any state S might be changed to any other state S'.

This is the foundation of set-based bidirectional transformations.

Important, pioneering work on set-based bidirectional transformations was carried out in this community by Hoffmann, Pierce et al, and by Stevens et al, among others.

1.2 Category-based bidirectional transformations

Other workers chose to analyse the state spaces differently.

Pioneering work by Diskin [4] et al noted that while the states are vitally important (and on one analysis, that of the database user, they are all that really matters) one might expect very different side-effects when one updates from S to S' in very different ways, and so it might be important to distinguish different updates leading from S to S'.

For example, S' might be obtainable from S by inserting a single row in a single table. Let's call that transition α. But there are many other ways to transition from S to S'. To take an extreme example, let's call it β, one might in a single update delete all of the contents of the database state S, and then insert into the resultant empty database all of the contents of the database state S'. Both α and β are transitions from S to S', so if one wishes to distinguish them, then a set-based, or indeed a co-discrete category based, description of the state space will not suffice. While the states themselves remain pre-eminently important, the transitions need to be tracked in detail, along with their compositions (one transition followed by another) and the state space becomes a category (we assume that the reader is familiar not just with set theory, but also with category theory).

Incidentally, much of the former controversy surrounding put-put laws arises from the different analyses of set-based and category-based bidirectional transformations. If a state space does not distinguish α from β then the result of maintaining consistency with α (a single row insert) has to be the same as the result of maintaining consistency with β and the latter could, on breaking down the update result in synchronising with the empty database state, and then synchronising with the update from that state to S'. On that analysis the put-put law (which says that an update can be synchronized only at the end, or at any intermediate step and then resynchronized at the end, with the same outcome in either case) is a very strong, probably unreasonably strong, requirement. Alternatively, if α and β are different updates then the put-put law says nothing *a priori* about their synchronizations and is a much less stringent requirement.

1.3 Information-order-based bidirectional transformations

There is yet another way in which one might analyse the state space of a relational database.

A basic transition might be inserting one or more new rows in some table(s). So we might consider a state-space with the same snapshots as before, but with an arrow $S \longrightarrow S'$ just when S' is obtained from S by inserting

rows. This is in fact the "information order" — the arrow $S \longrightarrow S'$ can be thought of as the inclusion of the snapshot S in the bigger (more information stored) snapshot S'.

Notice that this state space has arrows corresponding to inserts, and we will below call the inserts "rightwards" transitions. But those very same arrows correspond to deletes if we transition them "leftwards", backwards, along the arrow (one can move from S' to S by deleting the relevant rows).

The information order provides yet another potential state space for a relational database, and is well-understood by database theorists. It has the added complication that arrows can be transitioned in both directions. But it has the advantage of separating out two distinct kinds of transition, the rightwards, insert, transitions, and leftwards, delete, transitions, and these can be analysed separately.

It is of great convenience that transitions of the same kind — all leftwards or all rightwards — have very good properties: For example, an insert followed by another insert is definitely just an insert, and the corresponding "monotonic" (rightwards) put-put law for a Bx using multiple inserts has never been controversial. Similarly for multiple leftwards transitions.

Of course an arbitrary update can involve some mixture of leftwards and rightwards transitions, and the main technical content of this paper is the development of a detailed calculus for mixed transitions.

1.4 Bx state spaces

We have seen that there are (at least) three fundamental approaches to state spaces for relational databases. These approaches apply more generally to various sources for bidirectional transformations.

1. In many systems we can concentrate on the states, assume that we can transition from any state S to any other state S', and view the state space either as a set over which we might build a set-based Bx (for example a well-behaved lens), or equivalently as a co-discrete category (which explicitly says that there is a single arrow $S \longrightarrow S'$ for any two states S and S').

2. Alternatively, we can attempt to distinguish among different ways of updating $S \longrightarrow S'$, different "deltas", and view the state space as a category over which we might build a category-based Bx (for example, a delta-lens).

3. And very often among the categories of state spaces there is a "preferred direction" for arrows that can be identified in which case the state space as a category can be simplified by showing only the preferred direction, with arbitrary transitions recovered as composites of rightwards (normal direction along an arrow) and leftwards (backwards along an arrow) transitions.

Naturally, if a particular application lends itself well to set-based analysis then a set-based Bx will suffice.

Frequently however, knowledge of the deltas, when available, is sufficiently advantageous for us to want to build a category-based Bx. One advantage of the co-discrete representation of set-based bidirectional transformations is that set-based results can be derived from category-based results since co-discrete categories are merely a special case of categories.

Furthermore, if the application presents a natural "preferred" order of transition then the third approach might be used. Such information order and related situations are so common that this approach has been used implicitly for some time. The main goal of this paper is to develop the machinery that mathematically underlies this approach, and to show how it incorporates the other two approaches so that ordinary category-based, or indeed, via co-discrete categories, ordinary set-based bidirectional transformations, can be recovered as special cases.

1.5 Lenses as algebras for monads

In earlier work [13, 14, 9] the authors, sometimes with their colleague Wood, have shown that asymmetric lenses of various kinds are simply algebras for certain well-understood monads. This gives a strong, unified, and algebraic treatment of lenses, and brings to bear a wide-range of highly-developed mathematical tools.

The monads involved are all in some sense state space dependent.

For asymmetric lenses we will call the state spaces of the two systems $\mathbf{S}$ and $\mathbf{V}$, and suppose given a Get function or functor $G : \mathbf{S} \longrightarrow \mathbf{V}$.

In the set-based case the monad, $\Delta\Sigma$, captures completely the notion that all that is needed for a Put operation is a given state $S \in \mathbf{S}$ and a new state $V' \in \mathbf{V}$ with no necessary relationship between GS and V'.

In the category-based case the monad $(-, 1_V)$ (which we will rename here as R because it will be used to model the rightwards transitions), captures completely the notion that a Put operation depends on a given state $S \in \mathbf{S}$ and an arrow of the state space $\mathbf{V}$ of the form $GS \longrightarrow\!\!\!\!\!\rightarrow V'$.

In this paper we will show how to construct analogously a monad L which models the leftwards transitions, and which has as algebras lenses for the backwards (in database terms, delete) updates. A Put operation for leftwards transitions depends on a given state $S \in \mathbf{S}$ and an arrow of the state space $\mathbf{V}$ of the form $V' \longrightarrow\!\!\!\!\!\rightarrow GS$. (Notice that the update from GS to V' transitions *leftwards* along the arrow to reach V'.)

Most importantly in this paper we exhibit a distributive law, in the sense of Beck [3], which relates R and L and provides a new monad RL which captures completely the notion that for the general third case a Put operation should depend on a given state $S \in \mathbf{S}$ and an arbitrary composition of leftwards and rightwards arrows starting from GS and ending, after zig-zagging, at some $V' \in \mathbf{V}$. This is our main technical result.

Algebras for the new monad RL are lenses that synchronise with arbitrary strings of leftwards and rightwards transitions in $\mathbf{V}$.

1.6 Plan of this paper's technical content

The paper shows that the category theoretic structure of arrows rightwards from GS can be captured as a monad R. Similarly, the category theoretic structure of arrows leftwards from GS can be captured as a monad L. There are some important new technical advantages in this, and some important new implications for the lens community, but the basic ideas of these two monads are not new (first appearing in the paper of Street [17]).

Database theorists often work with the information order. But an arbitrary database transition involves inserts *and* deletes — a mixing of L and R transitions in the information order. Until now this has been done with handwaving — the R monad tells us all about inserts, the L monad tells us all about deletes, so we mix transitions together doing say a delete followed by an insert as a general transition which could be drawn as a "span" $S \longleftarrow S' \longrightarrow\!\!\!\!\!\rightarrow S''$ (get from S to S'' by deleting some rows from S to get to S', and then inserting some rows into S' to get to S''). And of course we want to distinguish this from other ways of getting from S to S'', perhaps $S \longleftarrow T \longrightarrow\!\!\!\!\!\rightarrow S''$ where T may be very different from S'. But we've moved from the monads that tell us everything we need to know about rightwards transitions and their interactions, and leftwards transitions and their interactions, to a vague idea of mixing such transitions together.

The main point of the paper is the discovery of a previously unobserved distributive law between L and R which, like all distributive laws, gives a new monad RL, and this composite monad precisely captures the calculus of these spans ("calculus" meaning we can calculate with it using routine procedures (various versions of μ and η below) determining algorithmically how all possible mixings, zigzags, etc, interact, which ones are equivalent to one another, and so on).

It should perhaps be noted here that the spans in this paper are mathematically the same as, but largely semantically otherwise unrelated to, the authors' use of equivalence classes of spans to describe *symmetric* lenses of various kinds in [10, 11].

The plan of this paper is fairly straightforward. In Section 2 we introduce in detail the two monads R and L. In Section 3 we develop the distributive law, slightly generalising the work of Beck as it is in fact a pseudo-distributive-law. (Recall that in category theory, when an axiom is weakened by requiring only a coherent isomorphism rather than an equality, the resulting notion is given the prefix "pseudo-". The coherency of the isomorphisms involved can be daunting, but in this paper the isomorphisms arise from universal properties and thus coherency is automatic and we will say no more about it.) In addition in Section 3 we display explicitly the basic operations of the new monad RL. In Section 4 we study the algebras for such monads, noting that these algebras are (generalised) lenses. And how useful might all this be? Well, Proposition 3 in Section 4 is an example of a strong new result that couldn't even be stated without the new monad RL.

We hope that having these things in mind might be some help in seeing through the technical details in the mathematics that follows.

1.7 Summarising the introduction

State spaces with reversible transitions have been the source of a number of confusions. The fact that every state is accessible from every other state in the same connected component, has sometimes led to researchers ignoring transitions (the set-based state spaces referred to above). Conversely, if instead of ignoring transitions *all* the transitions are explicitly included in the state space, then in many applications we are failing to distinguish two

different types of transition, the "rightwards" and the "leftwards". The resulting plethora of transitions of mixed types can seriously complicate any analysis.

With the two monads L and R, state spaces with reversible transitions can be managed effectively. Rather than constructing state spaces in which transitions come in pairs $S \longrightarrow S'$ and $S' \longrightarrow S$, we include only one of each pair (there is usually a "preferred" direction which can be the one included). Then the monad R is used for analyses and constructions using the categorical structure of the preferred transitions. Similarly the monad L can be used for analyses and constructions using the reverse transitions. And arbitrary composites of transitions can be broken down into "zig-zags" of L and R transitions, and in many cases into *spans*, L transitions followed by R transitions.

However, at this point we have come to the "hand-waving". What does it really mean mathematically to deal with zig-zags of L and R transitions? What does it mean to deal with an L transition followed by an R transition? And what is the calculus of mixed L and R transitions?

The main point of this paper is to show how under a modest hypothesis (the existence of pullbacks in the state space category $\mathbf{V}$) there is, up to isomorphism, a distributive law [3] between the two comma category monads R and L. In the presence of such a distributive law, the composite RL becomes a monad — the monad of anchored spans. The monad of anchored spans, including its relationship to L and R and the calculus it generates, answers all the questions in the preceding paragraph in a precise way. It eliminates the "hand-waving" and replaces it with a proper mathematical treatment of the interactions of R and L.

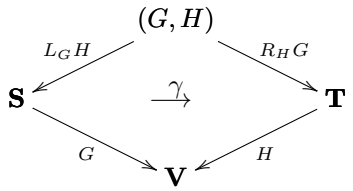
2 Two comma category monads

For basic category theoretic notions readers are referred to any of the standard texts, including for example [2] and [16]. At present the mathematical parts of this paper have been written succinctly, frequently assuming that readers have well-developed category theoretic backgrounds.

Given two functors with common codomain $\mathbf{V}$, say $\mathbf{S} \xrightarrow{G} \mathbf{V} \xleftarrow{H} \mathbf{T}$, the comma category (G, H) was introduced in the thesis of F.W. Lawvere. It has as objects triples (s, t, a) where s is an object of $\mathbf{S}$, t is an object of $\mathbf{T}$, and $a : Gs \longrightarrow Ht$ is an arrow of $\mathbf{V}$. The arrows of the comma category are, as one would expect, given by an arrow $p : s \longrightarrow s'$ of $\mathbf{S}$ and an arrow $q : t \longrightarrow t'$ of $\mathbf{T}$ which make the corresponding square in $\mathbf{V}$ commute (so for an arrow (p, q) from (s, t, a) to (s', t', a') we have $(Hq)a = a'(Gp)$ in $\mathbf{V}$) .

The comma category has evident projections to $\mathbf{S}$ and $\mathbf{T}$ shown in the figure below.

We denote the comma category and its projections as follows:



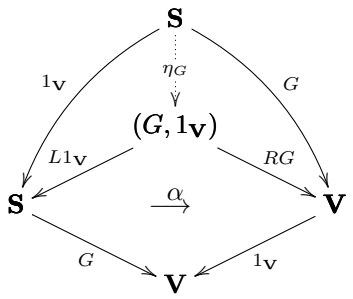
Where possible we will suppress the subscripts on projections. This is especially desirable if the subscript is itself an identity functor, and this situation arises frequently since the comma categories we will consider below will usually have the identity functor on $\mathbf{V}$ for one of either G or H.

The central arrow, γ in the figure, is included because the comma category has not just projections LH and RG, but also a natural transformation $\gamma : G(LH) \longrightarrow H(RG)$ (because each object (s, t, a) of (G, H) is in natural correspondence with an appropriate arrow, a itself, of $\mathbf{V}$). Indeed, the comma category is a kind of 2-categorical limit — it is universal among spans from $\mathbf{S}$ to $\mathbf{T}$ with an inscribed natural transformation.

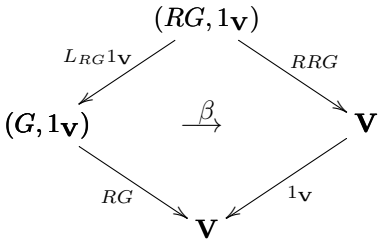
Explicitly, the universality of the comma category is given by the following. Each functor $F : \mathbf{X} \longrightarrow (G, H)$ corresponds bijectively with a triple (K, L, φ) where $K : \mathbf{X} \longrightarrow \mathbf{S}$, $L : \mathbf{X} \longrightarrow \mathbf{T}$ and $\varphi : GK \longrightarrow HL$ (with the correspondence given by composing each of LH, RG and γ with F).

Using this universal propertly we establish some further notation. Write η_G for the functor corresponding to

the triple $(1_{\mathbf{V}}, G, 1_G) : \mathbf{S} \longrightarrow (G, 1_{\mathbf{V}})$ as in



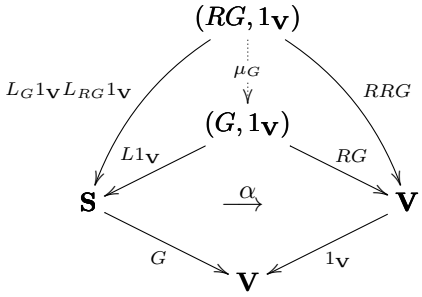
We denote the iterated comma category and projections



Write $\mu_G : (RG, 1_{\mathbf{V}}) \longrightarrow (G, 1_{\mathbf{V}})$, for the functor corresponding to the triple $(L_G 1_{\mathbf{V}} \cdot L_{RG} 1_{\mathbf{V}}, RRG, \beta(\alpha L_{RG} 1_{\mathbf{V}}))$, noting that the transformation is

$$G \cdot L_G 1_{\mathbf{V}} \cdot L_{RG} 1_{\mathbf{V}} \xrightarrow{\alpha L_{RG} 1_{\mathbf{V}}} RGL_{RG} 1_{\mathbf{V}} \xrightarrow{\beta} RRG$$

in



The assignment $G \mapsto RG$ (remembering that RG is more completely $R_{1_{\mathbf{V}}} G$, but as noted above we frequently suppress such subscripts) defines (on objects) the functor part of a monad on the slice category $\mathbf{cat}/\mathbf{V}$. The η_G and μ_G defined above are the unit and multiplication for this monad at an object G. The action of the functor on arrows, and the associativity and identity axioms for the monad, all follow from the facts that η and μ are defined by universal properties.

Similarly, $H \mapsto LH = L_{1_{\mathbf{V}}} H$ defines a monad L on $\mathbf{cat}/\mathbf{V}$.

3 The distributive law

For the remainder of this paper we assume that $\mathbf{V}$ has pullbacks.

Consider the monads R and L defined at the end of the previous section. We will denote the units and multiplications for R and L by η^R, η^L, μ^R and μ^L.

As we will show, there is a distributive law

$$LR \xrightarrow{\lambda} RL$$

between these two monads and consequently, the composite RL is a monad.

We begin by describing the composites LR and RL.

If $G : \mathbf{S} \longrightarrow \mathbf{V}$ then the domain of the functor RG is the comma category $(G, 1_{\mathbf{V}})$ and so has as objects arrows from $\mathbf{V}$ indexed by objects of $\mathbf{S}$ of the form $Gs \xrightarrow{a} v$. The important projection RG (important because it is the one which shows us the effect on G of the monad functor R) gives $RG(Gs \xrightarrow{a} v) = v$. The projection RG is also important because when we apply L to RG, L will build on the projected value v. Thus applying the functor L to RG gives a functor $LRG : (1_{\mathbf{V}}, RG) \longrightarrow \mathbf{V}$ whose domain has objects of the form $Gs \xrightarrow{a} v \xleftarrow{b} v'$, that is cospans (a, b) from Gs to v'. Moreover $LR(G)(a, b) = v'$.

On the other hand, the domain of the functor $LG : (1_{\mathbf{V}}, G) \longrightarrow \mathbf{V}$ has objects of the form $w \xrightarrow{c} Gs$ in $\mathbf{V}$ and $LG(w \xrightarrow{c} Gs) = w$. Applying the functor R to LG gives a functor $RL(G) : (LG, 1_{\mathbf{V}}) \longrightarrow \mathbf{V}$ whose domain has objects of the form $Gs \xleftarrow{c} w \xrightarrow{d} w'$, that is spans (c, d) from Gs to w', and $RL(G)(c, d) = w'$.

Now we can define the G'th component of λ. It is the (pseudo-)functor (over $\mathbf{V}$) $\lambda_G : (1_{\mathbf{V}}, RG) \longrightarrow (LG, 1_{\mathbf{V}})$ defined at an object $Gs \xrightarrow{a} v \xleftarrow{b} v'$ of the domain of LRG by taking the pullback in $\mathbf{V}$ (and on arrows using the induced maps). Thus $\lambda_G(a, b)$ is a span in $\mathbf{V}$ from Gs to v'. The value of the functor $LR(G)$ at (a, b) is v' and the value of $RL(G)$ at the pullback of the cospan (a, b) is also v'. Thus, $RL(G)(\lambda_G(a, b)) = LR(G)(a, b)$ on the nose.

To see that λ is natural, suppose that $G' : \mathbf{S}' \longrightarrow \mathbf{V}$ and the functor $F : \mathbf{S} \longrightarrow \mathbf{S}'$ defines an arrow from G to G' in $\mathbf{cat}/\mathbf{V}$, that is $G'F = G$. We need to show that $\lambda_{G'} LR(F) = RL(F)\lambda_G$. Thus the following square of functors must commute (over $\mathbf{V}$):

$$\begin{array}{ccc} (1_{\mathbf{V}}, RG) & \xrightarrow{\lambda_G} & (LG, 1_{\mathbf{V}}) \\ {\scriptstyle LR(F)}\downarrow & & \downarrow{\scriptstyle RL(F)} \\ (1_{\mathbf{V}}, RG') & \xrightarrow[\lambda_{G'}]{} & (LG', 1_{\mathbf{V}}) \end{array}$$

where $LR(F)$ is the functor $(1_{\mathbf{V}}, R(F))$ which defines a morphism $LRG \longrightarrow LRG'$. The square does commute, up to isomorphism, since the effect of $LR(F)$ on a cospan $Gs \xrightarrow{a} v \xleftarrow{b} v'$ (an object of $(1_{\mathbf{V}}, RG)$) is actually the same cospan $G'Fs = Gs \xrightarrow{a} v \xleftarrow{b} v'$ and $\lambda_{G'}$ computes a pullback span, $G'Fs \xleftarrow{c} w \xrightarrow{d} v'$. On the other hand $RL(F)\lambda_G$ applied to the (same) cospan (a, b) computes a pullback span, $Gs \xleftarrow{c'} w' \xrightarrow{d'} v'$, and then applies $RL(F)$ which likewise leaves the span unchanged, and so isomorphic to (c, d).

Next we consider the distributive law equations [3]. One equation involving the units is:

$$\begin{array}{ccc} & L & \\ {\scriptstyle L\eta^R}\swarrow & & \searrow{\scriptstyle \eta^R L} \\ LR & \xrightarrow[\lambda]{} & RL \end{array}$$

At a functor G, the left hand side of the triangle $L\eta^R$ applies to an object $v \xrightarrow{a} Gs$ of the domain of LG and the result is the cospan $v \xrightarrow{a} Gs \xleftarrow{1} Gs$. Application of λ_G gives a pullback span which we can choose to be $v \xleftarrow{1} v \xrightarrow{a} Gs$. On the other hand $\eta^R L$ applied to $v \xrightarrow{a} Gs$ is exactly the same span $v \xleftarrow{1} v \xrightarrow{a} Gs$.

The other unit equation is:

$$\begin{array}{ccc} & R & \\ {\scriptstyle \eta^L R}\swarrow & & \searrow{\scriptstyle R\eta^L} \\ LR & \xrightarrow[\lambda]{} & RL \end{array}$$

At a functor G, the left hand side $\eta^L R$ applies to an object $Gs \xrightarrow{b} v$ of the domain of RG and the result is the cospan $Gs \xrightarrow{b} v \xleftarrow{1} v$. Application of λ_G gives a pullback span which we choose to be $Gs \xleftarrow{1} Gs \xrightarrow{b} v$. Again, this is the same as the effect of $R\eta^L$ on $Gs \xrightarrow{b} v$.

We consider only one of the equations involving multiplications; the other is similar. The equation we consider is:

$$\begin{array}{ccccc} LRR & \xrightarrow{\lambda R} & RLR & \xrightarrow{R\lambda} & RRL \\ {\scriptstyle L\mu^R}\downarrow & & & & \downarrow{\scriptstyle \mu^R L} \\ LR & & \xrightarrow[\lambda]{} & & RL \end{array}$$

Again, we look at the equation in the domain and at an object G. A typical object of the domain of $LRR(G)$ is an extended cospan of the form $Gs \xrightarrow{a} v \xrightarrow{a'} v' \xleftarrow{b} w$. Since $\mu^R(a, a')$ is simply the composite $Gs \xrightarrow{a'a} v'$, we see that $\lambda_G(L\mu^R(G))(a, a', b)$ is a pullback span $Gs \xleftarrow{c} u \xrightarrow{d} w$ of b along $a'a$. On the other hand, $\lambda_G R(G)$ applied to (a, a', b) computes a pullback span of b along a' with result $Gs \xrightarrow{a} v \xleftarrow{c'} u' \xrightarrow{d'} w$. Applying $RG\lambda_G$ computes a pullback of c' along a with result $Gs \xleftarrow{c''} u'' \xrightarrow{d''} u' \xrightarrow{d'} w$. Applying $\mu^R_G LG$ composes d'' and d' giving the span $Gs \xleftarrow{c''} u'' \xrightarrow{d''d'} w$ which is isomorphic to $Gs \xleftarrow{c} u \xrightarrow{d} w$.

The preceding considerations give:

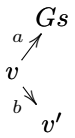
Proposition 1 *The transformation $LR \xrightarrow{\lambda} RL$ is a (pseudo-)distributive law.* ∎

Using this proposition, and with minor modifications of the work of Beck [3] to take account of the isomorphisms in the pseudo-naturality squares and in the equations involving multiplications, we obtain:

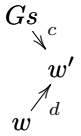
Proposition 2 *The composite functor RL on $\mathbf{cat}/\mathbf{V}$ is a monad, the monad of spans, with $\mu^{RL} = \mu^R L \cdot RR\mu^L \cdot R\lambda L$ and $\eta^{RL} = \eta^R L \cdot \eta^L$.* ∎

It is convenient for later work to introduce some notation and use it to describe the unit and multiplication for the composed monad RL.

As noted above, for $G : \mathbf{S} \longrightarrow \mathbf{V}$, the domain of RLG is a comma category whose objects are certain spans in $\mathbf{V}$. Let us denote them

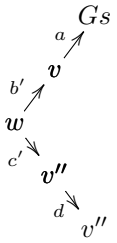


Objects of the domain of LRG are depicted:



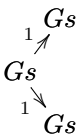
An object of the domain of $RLRLG$ is a "zig-zag":

The effect of the multiplication for RL on the zig-zag is to first form the span:



where w is a pullback of b and c, and then to compose each of the legs using the multiplication μ^L for the top leg (which is an instance of LLG), and the multiplication μ^R for the bottom leg (which is by then an instance of $RRLG$).

The unit for RL simply forms spans of identity arrows, so each object s of $\mathbf{S}$ yields a span



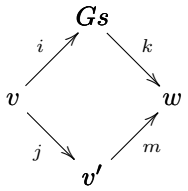
4 Algebras

To illustrate the benefits of the distributive law and the resultant monad RL we present here a significantly stronger version, made possible by the use of RL, of a result from [8]. That paper was a study of two kinds of generalised lenses, one for preferred transitions and one for their reverses. It showed that the two lenses are respectively an R-algebra and an L-algebra and conversely. The main interest was how they should interact via a "mixed put-put law" called condition (*) below. The new result shows that having two such lenses satisfying condition (*) is equivalent to having an RL-algebra. In other words, if we can update inserts (R transitions) and deletes (L transitions) and those updates satisfy condition (*), then we can update spans — arbitrary compositions of R and L transitions.

We first describe a condition that is essentially the Beck-Chevalley condition for our purposes. (It does not matter if the reader is not familiar with other instances of the Beck-Chevalley condition.)

Suppose that $\mathbf{S} \xrightarrow{G} \mathbf{V}$ and that $RG \xrightarrow{r} G$ and $LG \xrightarrow{l} G$ are R- and L-algebra structures. If $Gs \xrightarrow{k} w$ is an object of $(G, 1_{\mathbf{V}})$, we denote its image under r by $r(s,k)$. Similarly if $v \xrightarrow{i} Gs$ is an object of $(1_{\mathbf{V}}, G)$, we denote its image under l by $l(i,s)$. Since r and l are arrows in $\mathbf{cat}/\mathbf{V}$, $G(r(s,k)) = w$ and $G(l(i,s)) = v$. We say that *condition (*) is satisfied* if

for any object s in $\mathbf{S}$ and any pullback (in $\mathbf{V}$)



it is the case that $r(l(i,s),j) \cong l(m, r(s,k))$.

Proposition 3 *If $RLG \xrightarrow{\xi} G$ is an RL-algebra, then $r = \xi R\eta^L G$ and $l = \xi\eta^R LG$ define R- and L-algebras, respectively, that satisfy condition (*). Conversely, suppose $RG \xrightarrow{r} G$ and $LG \xrightarrow{l} G$ are R- and L-algebra structures satisfying (*). Define $\xi : RLG \longrightarrow G$ (on objects) by $\xi(Gs \xleftarrow{i} v \xrightarrow{j} v') = r(l(i,s),j)$. Then ξ determines an RL-algebra structure on G.*

Proof. Suppose $RLG \xrightarrow{\xi} G$ is an RL-algebra, r and l are defined as in the statement, and the square in condition (*) is a pullback in $\mathbf{V}$. The definitions of r and l amount, on objects, to $r(Gs \xrightarrow{a} v) = \xi(Gs \xleftarrow{1} Gs \xrightarrow{a} v)$ and $l(v \xrightarrow{a} Gs) = \xi(Gs \xleftarrow{a} v \xrightarrow{1} v)$. For clarity we will sometimes suppress the objects of $\mathbf{V}$, (v, Gs, etc), in what follows. Then, using the notation from the pullback square and noting that $G(\xi(\xleftarrow{1} \xrightarrow{k})) = w$,

$$\begin{aligned}
l(m, r(s,k)) &= l(m, \xi(\xleftarrow{1} \xrightarrow{k})) \\
&= \xi(G(\xi(\xleftarrow{1} \xrightarrow{k})) \xleftarrow{m} \xrightarrow{1}) \\
&= \xi \cdot RL(\xi)(\xleftarrow{1} \xrightarrow{k} \xleftarrow{m} \xrightarrow{1}) \\
&= \xi\mu_G(\xleftarrow{1} \xrightarrow{k} \xleftarrow{m} \xrightarrow{1}) \\
&= \xi(\xleftarrow{1} \xleftarrow{i} \xrightarrow{j} \xrightarrow{1}) \\
&= \xi(\xleftarrow{i} \xrightarrow{j}) \\
&= r(\xi(\xleftarrow{i} \xrightarrow{1}), j) \\
&= r(l(i,s), j)
\end{aligned}$$

(where the fourth equality uses the associative law for the RL-algebra ξ) which proves condition (*).

To check that r and l satisfy the algebra axioms for R and L respectively is routine.

Conversely, suppose that r and l are R- and L-algebras respectively, and that they satisfy condition (*). Suppose that ξ is defined on objects as in the statement. Notice that it is straightforward to extend the definition of ξ to arrows of (the domain of) RLG.

For the RL-algebra associative law we need to verify that for a "zig-zag" $W = Gs \xleftarrow{x} \xrightarrow{y} \xleftarrow{z} \xrightarrow{w}$ starting from Gs, say, that $\xi RL\xi(W) = \xi\mu_G(W)$. Now $RL(\xi)$ applies ξ to $\xleftarrow{x} \xrightarrow{y}$ (while carrying along z and w) giving an object G-over the codomain of z, and ξ applies to this result along with z and w. So using the definition of ξ above, and writing $\xleftarrow{a} \xrightarrow{b}$ for the pullback of $\xrightarrow{y} \xleftarrow{z}$,

$$\begin{aligned}
\xi RL(\xi)(W) &= \xi(G(r(l(x,s),y)) \xleftarrow{z} \xrightarrow{w}) \\
&= r(l(z, r(l(x,s),y)), w) \\
&= r(r(l(a, l(x,s)), b), w) \\
&= r(r(l(xa, s), b), w) \\
&= r(l(xa, s), wb) \\
&= \xi(\xleftarrow{xa} \xrightarrow{wb}) \\
&= \xi(\mu_G(W))
\end{aligned}$$

in which the third equation is an application of property (*) and the fourth and fifth equations use the associative laws of the L-algebra and the R-algebra respectively.

The RL-algebra identity law for ξ is straightforward noting the definition of η^{RL} in terms of η^R and η^L. ∎

5 Related work

In the development of a 2-categorical treatment of the Yoneda Lemma, Ross Street [17] studied monads equivalent to L and R, and also a 'composite' monad M, *not* equivalent to the composite studied here. The present authors, and their colleague Wood, introduced L and R as part of an on-going study of the use of monads in the study of generalised lenses [14]. That work, like much of the earlier work of Johnson and Rosebrugh starting from [6], studied inserts and deletes in isolation, and depended upon the presumption that these could then be reintegrated as spans. A theoretical 2-categorical analysis of spans of models was carried out in [12], but it has had little practical application to date. More recently, a search for a 'mixed put-put law' [8] led the authors to the discovery

of the distributive law reported here, and hence to the new composite monad RL and the corresponding calculus of mixed transitions.

In the realm of database state spaces it seems that most authors have taken the set-based state space approach. We have argued elsewhere [7] that view updating has been limited unnecessarily to constant complement updating [1] because of the failure to treat transitions as first class citizens. One notable exception is the insightful work of Hegner [5] which introduced an information order on the set of states. This order corresponds precisely to choosing to include insert transitions in the state space, but to leave delete transitions out (to be recovered as reversed insert transitions as described in Section 1 above).

Spans have a wide variety of applications, and so have long been studied category theoretically. Given a category $\mathbf{C}$ with pullbacks there is a bicategory $\mathsf{span}\mathbf{C}$ with the same objects as $\mathbf{C}$, with spans of arrows from $\mathbf{C}$ as arrows, and with composition given by pullback, and most treatments take this point of view. Such spans are oriented, despite their symmetry, by definition, since they are arrows of a bicategory. To the authors' knowledge, the monad of anchored spans presented here, including its relationship to the comma category monads L and R, is entirely new. In addition it has noteworthy and desirable differences from earlier treatments because the orientation of spans comes from the fibering over $\mathbf{V}$, and because, in the manner of comma categories, spans appear as objects rather than as arrows.

Meanwhile spans have also had a wide range of applications among researches into bidirectional transformations. In particular the graph transformation community has used spans for many years and the recent paper of Orejas et al [15] makes use of spans of updates in a manner closely related to the current paper. We leave to future work the exploration of the possible applications of our developments in those areas.

6 Conclusions

The main findings from this paper are

1. By explicitly working on $\mathbf{cat}/\mathbf{V}$ the two comma categories $(G, 1_{\mathbf{V}})$ and $(1_{\mathbf{V}}, G)$ can be fibred over $\mathbf{V}$ and so sources and targets can be tracked, resulting in monads R and L respectively. Those monads capture the category theoretic structure of arrows out of images of G and into images of G respectively, and, being built from those comma categories, they lift arrows of $\mathbf{V}$ to objects of RG and LG (fibred over $\mathbf{V}$).

2. There is a pseudo-distributive law between R and L, and so RL is itself a monad, the monad of anchored spans. Furthermore the tracking of sources and targets, provided by the fibring, orients the spans as spans from images of G (the "anchoring"). The monad RL captures the category theoretic structure of spans from images of G, and, being built from iterated comma categories, lifts spans in $\mathbf{V}$ from images of G to objects of RLG (fibred over $\mathbf{V}$).

3. Having an RL-algebra is equivalent to having a pair of algebras (one R and one L) satisfying condition (*).

The first finding shows that we have found the right context in which to work with comma categories for a variety of state space based applications. The second finding solves the long standing problem of making mathematically precise the composition of preferred and reversed transitions (eliminating the "hand-waving"). The third finding neatly illustrates the extra power available when spans of transitions can be dealt with as single objects.

7 Acknowledgements

The authors are grateful for the support of the Australian Research Council and NSERC Canada, and for insightful suggestions from anonymous referees.

References

[1] Bancilhon, F. and Spyratos, N. (1981) Update Semantics of Relational Views, *ACM Trans. Database Syst.* **6**, 557–575.

[2] Barr, M. and Wells, C. (1995) *Category Theory for Computing Science.* Prentice-Hall.

[3] Beck, J. M. (1969) Distributive Laws. *Seminar on Triples and Categorical Homology Theory* Lecture Notes in Math. **80**, 95–112. Available in Reprints in *Theory Appl. Categ.* **18** (2008).

[4] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki (2011), From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case, *Journal of Object Technology* **10**, 6:1–25, doi:10.5381/jot.2011.10.1.a6

[5] Hegner, S. J. (2004) An Order-Based Theory of Updates for Closed Database Views. *Ann. Math. Artif. Intell.* **40**, 63–125.

[6] Johnson, M. and Rosebrugh, R. (2001) View Updatability Based on the Models of a Formal Specification. *Proceedings of Formal Methods Europe 2001*, Lecture Notes in Comp. Sci. **2021**, 534–549.

[7] Johnson, M. and Rosebrugh, R. (2007) Fibrations and Universal View Updatability. *Theoret. Comput. Sci.* **388**, 109–129.

[8] Johnson, M. and Rosebrugh, R. (2012) Lens Put-Put Laws: Monotonic and Mixed. *Electronic Communications of the EASST*, **49**, 13pp.

[9] Johnson, M. and Rosebrugh, R. (2013) Delta Lenses and Fibrations. *Electronic Communications of the EASST*, **57**, 18pp.

[10] Johnson, M. and Rosebrugh, R. (2014) Spans of Lenses. *CEUR Proceedings*, **1133**, 112–118.

[11] Johnson, M. and Rosebrugh, R. (2015) Spans of Delta Lenses. To appear *CEUR Proceedings*.

[12] Johnson, M., Rosebrugh, R. and Wood, R. J. (2002) Entity-Relationship-Attribute Designs and Sketches. *Theory Appl. Categ.* **10**, 94–112.

[13] Johnson, M., Rosebrugh, R. and Wood, R. J. (2010) Algebras and Update Strategies. *J.UCS* **16**, 729–748.

[14] Johnson, M., Rosebrugh, R. and Wood, R. J. (2012) Lenses, Fibrations, and Universal Translations. *Math. Structures in Comp. Sci.* **22**, 25–42

[15] Orejas, F., Boronat, A., Ehrig, H., Hermann, F., and Schölzel, H. (2013) On Propagation-Based Concurrent Model Synchronization. *Electronic Communications of the EASST* **57**, 19pp.

[16] Pierce, B. (1991) *Basic Category Theory for Computer Scientists*. MIT Press.

[17] Street, R. (1974) Fibrations and Yoneda's Lemma in a 2-category. *Lecture Notes in Math.* **420**, 104–133

BiYacc: Roll Your Parser and Reflective Printer into One

Zirun Zhu[1,2] Hsiang-Shang Ko[2] Pedro Martins[3] João Saraiva[3] Zhenjiang Hu[1,2]

[1] SOKENDAI (The Graduate University for Advanced Studies), Japan
{zhu,hu}@nii.ac.jp
[2] National Institute of Informatics, Japan
hsiang-shang@nii.ac.jp
[3] HASLab/INESC TEC & University of Minho, Portugal
{prmartins,jas}@di.uminho.pt

Abstract

Language designers usually need to implement parsers and printers. Despite being two related programs, in practice they are designed and implemented separately. This approach has an obvious disadvantage: as a language evolves, both its parser and printer need to be separately revised and kept synchronised. Such tasks are routine but complicated and error-prone. To facilitate these tasks, we propose a language called BiYacc, whose programs denote both a parser and a printer. In essence, BiYacc is a domain-specific language for writing *putback-based* bidirectional transformations — the printer is a putback transformation, and the parser is the corresponding get transformation. The pairs of parsers and printers generated by BiYacc are thus always guaranteed to satisfy the usual round-trip properties. The highlight that distinguishes this *reflective* printer from others is that the printer — being a putback transformation — accepts not only an abstract syntax tree but also a string, and produces an updated string consistent with the given abstract syntax tree. We can thus make use of the additional input string, with mechanisms such as simultaneous pattern matching on the view and the source, to provide users with full control over the printing-strategies.

1 Introduction

Whenever we come up with a new programming language, as part of its compiler we need to design and implement a parser and a printer to convert between program text and its internal representation. A piece of program text, while conforming to a *concrete syntax* specification, is a flat string that can be easily edited by the programmer. The parser recovers the tree structure of such a string and converts it to an *abstract syntax* tree, which is a structured and simplified representation that is easier for the compiler backend to manipulate. On the other hand, a printer flattens an abstract syntax tree to a string, which is typically in a human readable format. This is useful for debugging the compiler or reporting internal information to the user, for example.

Parsers and printers do conversions in opposite directions, however, they are closely related — for example, we normally expect that a string printed from an abstract syntax tree can be parsed to the same tree. This is also clearly shown on language-based editors, as introduced by Reps [20, 21], where the user interacts with a pretty printed representation of the underlying abstract syntax tree. Thus, each user text update is performed as an

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

abstract syntax tree transformation that has to be automatically synchronised with its concrete representation. Despite this relationship, current practice is to write parsers and printers separately. This approach has an obvious disadvantage: the parser and the printer need to be revised from time to time as the language evolves. Each time, we must revise the parser and the printer and also keep them consistent with each other, which is a time-consuming and error-prone task. To address the problem, we propose a prototype domain-specific language BiYacc, in which the user can describe both a parser and a printer in a single program, contrary to designing and writing them separately as is traditional. By unifying these two pieces of software and deriving them from single, unambiguous and centralised code, we are creating a unified environment, which is easier to maintain and update, therefore respecting the "Don't Repeat Yourself" principle of software development [11].

Distinct from traditional kinds of printers, the printer generated from a BiYacc program is reflective: it takes not only an abstract syntax tree but also a piece of program text, and produces an updated piece of program text into which information from the abstract syntax tree is properly embedded. We illustrate reflective printing with the following small but non-trivial example (which is also used as the running example in subsequent sections) about a simple language of arithmetic expressions. The concrete syntax has negation, parentheses, and the four elementary arithmetic operations, while the abstract syntax has only the four elementary arithmetic operations — negated expressions are represented in the abstract syntax as subtraction expressions whose left operand is zero, and parenthesised expressions are translated into tree structures. Now suppose we write an arithmetic expression as a plain string, and parse it to an abstract syntax tree. Later, the abstract syntax tree is somehow modified (say, after some optimisation done by the compiler), and we want to print it back to a string for the user to compare with what was written originally (say, to understand what the compiler's optimisation does). To make it easier for the user to compare these two strings, we should try to maintain the syntactic characteristics of the original string when producing the updated string. Here we may choose to

- preserve all brackets — even redundant ones — in the original string, and
- preserve negation expressions in the original string instead of changing them to subtraction expressions.

For example, the string "$((-1))$" is parsed to an abstract syntax tree "`SUB (NUM 0) (NUM 1)`" (a subtraction node whose left subtree is a numeral node `0` and right subtree is another numeral node `1`); if we change the abstract syntax tree to "`SUB (NUM 0) (NUM 2)`" and update the string with our reflective printer, we get "$((-2))$" instead of "$0 - 2$". (Section 2 will present a BiYacc program that describes exactly this printing strategy.) Reflective printing is a generalisation of traditional printing because our reflective printer can accept an abstract syntax tree and an *empty* string, in which case it will behave just like a traditional printer, producing a new string depending on the abstract syntax tree only.

Under the bonnet, BiYacc is based on the theory of *bidirectional transformations* (BXs for short) [2, 7, 9]. BXs are used for synchronising two sets of data, one called the *source* and the other the *view*. Denoting the source set by S and the view set by V, a (well-behaved) BX is a pair of functions called *get* and *put*:

- the function $get : S \to V$ extracts a part of a source of interest to the user as a *view*, while
- the function $put : S \times V \to S$ takes a source and a view and produces an updated source incorporating information from the view.

The pair of functions should satisfy the following laws:

$$get(put(s, v)) = v \qquad \forall s \in S,\ v \in V \tag{PutGet}$$

$$put(s, get(s)) = s \qquad \forall s \in S \tag{GetPut}$$

Informally, the PutGet law enforces that *put* must embed all information of the view into the updated source, so the view can be recovered from the source by *get*, while the GetPut law prohibits *put* from performing unnecessary updates by requiring that putting back a view directly extracted from a source by *get* must produce the same, unmodified source. The parser and reflective printer generated from a BiYacc program are exactly the functions *get* and *put* in a BX, and are thus guaranteed to satisfy the PutGet and GetPut laws. In the context of parsing and printing, PutGet ensures that a string printed from an abstract syntax tree is parsed to the same tree, and GetPut ensures that updating a string with an abstract syntax tree parsed from the string leaves the string unmodified (including formatting details like parentheses). A BiYacc program thus not only conveniently expresses a parser and a reflective printer simultaneously, but also ensures that the parser and the reflective printer are consistent with each other in a precise sense.

```
1  Abstract
2
3  Arith = ADD Arith Arith
4        | SUB Arith Arith
5        | MUL Arith Arith
6        | DIV Arith Arith
7        | NUM String
8
9  Concrete
10
11 Expr   -> Expr '+' Term
12         | Expr '-' Term
13         | Term
14
15 Term   -> Term '*' Factor
16         | Term '/' Factor
17         | Factor
18
19 Factor -> '-' Factor
20         | String
21         | '(' Expr ')'
22
23 Actions
24
25 Arith +> Expr
26 ADD x y  ->  (x => Expr) '+' (y => Term)
27 SUB x y  ->  (x => Expr) '-' (y => Term)
28 arith    ->  (arith => Term)
29
30 Arith +> Term
31 MUL x y  ->  (x => Term) '*' (y => Factor)
32 DIV x y  ->  (x => Term) '/' (y => Factor)
33 arith    ->  (arith => Factor)
34
35 Arith +> Factor
36 SUB (NUM "0") y  ->  '-' (y => Factor)
37 NUM n            ->  (n => String)
38 arith            ->  '(' (arith => Expr) ')'
```

Figure 1: A BiYacc program for the expression example.

We have implemented BiYacc in Haskell and tested the example about arithmetic expressions mentioned above, which we will go through in Section 2. There is an interactive demo website:

`http://www.prg.nii.ac.jp/project/biyacc.html`

from which the source code of BiYacc can also be downloaded. The reader is invited to vary the input source string and abstract syntax tree, run the forward and backward transformations, and even modify the BiYacc programs to see how the behaviour changes. A sketch of the implementation is presented in Section 3, and we conclude the paper with some discussions of related work in Section 4.

2 An overview of BiYacc

This section gives an overview to the structure, syntax, and semantics of BiYacc by going through a program dealing with the example about arithmetic expressions. The program, shown in Figure 1, consists of three parts:

- abstract syntax definition,
- concrete syntax definition, and
- actions describing how to update a concrete syntax tree with an abstract syntax tree.

2.1 Defining the abstract syntax

The abstract syntax part of a BiYacc program starts with the keyword `Abstract`, and can be seen as definitions of *algebraic datatypes* commonly found in functional programming languages (see, e.g., [10, Section 5]). For the expression example, we define a datatype `Arith` of arithmetic expressions, where an arithmetic expression can be either an addition, a subtraction, a multiplication, a division, or a numeric literal. For simplicity, we represent a literal as a string. Different constructors — namely `ADD`, `SUB`, `MUL`, `DIV`, and `NUM` — are used to construct different kinds of expressions, and in the definition each constructor is followed by the types of arguments it takes. Hence the constructors `ADD`, `SUB`, `MUL`, and `DIV` take two subexpressions (of type `Arith`) as arguments, while the last constructor `NUM` takes a `String` as argument. (`String` is a built-in datatype of strings.) For instance, the expression "$1 - 2 \times 3 + 4$" can be represented as this abstract syntax tree of type `Arith`:

```
ADD (SUB (NUM "1")
         (MUL (NUM "2")
              (NUM "3")))
    (NUM "4")
```

2.2 Defining the concrete syntax

Compared with an abstract syntax, the structure of a concrete syntax is more refined such that we can conveniently yet unambiguously represent a concrete syntax tree as a string. For instance, we should be able to interpret the string "$1 - 2 \times 3 + 4$" unambiguously as a tree of the same shape as the abstract syntax tree at the end of the previous subsection. This means that the concrete syntax for expressions should, in particular, somehow encode the conventions that multiplicative operators have higher precedence than additive operators and that operators of the same precedence associate to the left.

In a BiYacc program, the concrete syntax is defined in the second part starting with the keyword `Concrete`. The definition is in the form of a context-free grammar, which is a set of production rules specifying how nonterminal symbols can be expanded to sequences of terminal or nonterminal symbols. For the expression example, we use a standard syntactic structure to encode operator precedence and order of association, which involves three nonterminal symbols `Expr`, `Term`, and `Factor`: an `Expr` can produce a left-leaning tree of `Term`s, each of which can in turn produce a left-leaning tree of `Factor`s. To produce right-leaning trees or operators of lower precedence under those with higher precedence, the only way is to reach for the last production rule `Factor -> '(' Expr ')'`, resulting in parentheses in the produced string.

Note that there is one more difference between the concrete syntax and the abstract syntax in this example: the concrete syntax has a production rule `Factor -> '-' Factor` for producing negated expressions, whereas in the abstract syntax we can only write subtractions. This means that negative numbers will have to be converted to subtractions and these subtractions will have to be converted back to negative numbers in the opposite direction. As we will see, this mismatch can be easily handled in BiYacc.

2.3 Defining the actions

The last and main part of a BiYacc program starts with the keyword `Actions`, and describes how to update a concrete syntax tree — i.e., a well-formed string — with an abstract syntax tree. Note that we are identifying strings representing program text with concrete syntax trees: Conceptually, whenever we write an expression as a string, we are actually describing a concrete syntax tree with the string (instead of just describing a sequence of characters). Technically, it is almost effortless to convert a (well-formed) string to a concrete syntax tree with existing parser technologies; the reverse direction is even easier, requiring only a traversal of the concrete syntax tree. By integrating with existing parser technologies, BiYacc actions can focus on describing conversions between concrete and abstract syntax trees — the more interesting part in the tasks of parsing and pretty-printing.

2.3.1 Individual action groups

The `Actions` part consists of groups of actions, and each group of actions begins with a type declaration:

abstract syntax datatype `+>` *concrete nonterminal symbol*

The symbol '`+>`' indicates that this group of actions describe how to put information from an abstract syntax tree of the specified datatype into a concrete syntax tree produced from the specified nonterminal symbol. Each action takes the form

abstract syntax pattern `->` *concrete syntax update pattern*

The left-hand side pattern describes a particular shape for abstract syntax trees and the right-hand side one for concrete syntax trees; also the right-hand side pattern is overlaid with updating instructions denoted by '`=>`'. For brevity, we call the left-hand side patterns *view patterns* and the right-hand side ones *source patterns* (in this case, representing an abstract and a concrete representation, respectively), hinting at their roles in terms of the underlying theory of bidirectional transformations. Given an abstract syntax tree and a concrete syntax tree, the semantics of an action is to simultaneously perform *pattern matching* on both trees (like in functional programming languages), and then use components of the abstract syntax tree to update parts of the concrete syntax tree, possibly recursively.

2.3.2 Individual actions

Let us look at a specific action — the first one for the expression example, at line 26 of Figure 1:

```
ADD x y -> (x => Expr) '+' (y => Term)
```

For the view pattern `ADD x y`, an abstract syntax tree (of type `Arith`) is said to match the pattern when it starts with the constructor `ADD`; if the match succeeds, the two arguments of the constructor (i.e., the two subexpressions of the addition expression) are then respectively bound to the variables `x` and `y`. (BiYacc adopts the naming convention in which variable names start with a lowercase letter and names of datatypes and nonterminal symbols start with an uppercase letter.) For the source pattern of the action, the main intention is to refer to the production rule

```
Expr -> Expr '+' Term
```

and use this to match those concrete syntax trees produced by first using this rule. Since the action belongs to the group `Arith +> Expr`, the part '`Expr ->`' of the production rule can be inferred and hence is not included in the source pattern. Finally we overlay '`x =>`' and '`y =>`' on the nonterminal symbols `Expr` and `Term` to indicate that, after the simultaneous pattern matching succeeds, the subtrees `x` and `y` of the abstract syntax tree are respectively used to update the left and right subtrees of the concrete syntax tree.

It is interesting to note that more complex view and source patterns are also supported, which can greatly enhance the flexibility of actions. For example, the view pattern

```
SUB (NUM "0") y
```

of the action at line 36 of Figure 1 accepts those subtraction expressions whose left subexpression is zero. This action is the key to preserving negation expressions in the concrete syntax tree. For an example of a more complex source pattern: Suppose that in the `Arith +> Factor` group we want to write a pattern that matches those concrete syntax trees produced by the rule `Factor -> '-' Factor`, where the inner nonterminal `Factor` produces a further `'-' Factor` using the same rule. This pattern is written by overlaying the production rule on the nonterminal `Factor` in the top-level appearance of the rule:

```
'-' (Factor -> '-' Factor)
```

2.3.3 Semantics of the entire program

Now we can explain the semantics of the entire program. Given an abstract syntax tree and a concrete syntax tree as input, first a group of actions is chosen according to the types of the trees. Then the actions in the group are tried in order, by performing simultaneous pattern matching on both trees. If pattern matching for an action succeeds, the update specified by the action is executed (recursively); otherwise the next action is tried. (Execution of the program stops when the matched action specifies either no updating operations or only updates to `String`.) BiYacc's most interesting behaviour shows up when pattern matching for all of the actions in the chosen group fail: in this case a suitable source will be created. The specific approach here is to do pattern matching just on the abstract syntax tree and choose the first matching action. A suitable concrete syntax tree matching the source pattern is then created, whose subtrees are recursively created according to the abstract syntax tree. We justify this approach as follows: if none of the source patterns match, it means that the input

concrete syntax tree differs too much from the abstract syntax tree, so we should throw the concrete syntax tree away and print a new one according to the abstract syntax tree.

2.3.4 An example of program execution

To illustrate, let us go through the execution of the program in Figure 1 on the abstract syntax tree

```
ADD (SUB (NUM 0) (NUM 4)) (NUM 5)
```

and the concrete syntax tree denoted by the string

$(-1 + 2 * 3)$

The abstract syntax tree is obtained from the concrete syntax tree by ignoring the pair of parentheses, desugaring the negation to a subtraction, and replacing the number 1 with 4 and the multiplication subexpression with 5. Executing the program will leave the pair of parentheses intact, update the number 1 in the concrete syntax tree with 4, preserving the negation, and update the multiplication subexpression to 5. In detail:

1. Initially the types of the two trees are assumed to match those declared for the first group, and hence we try the first action in the group, at line 26. The view-side pattern matching succeeds but the source-side one fails, because the first production rule used for the source is not `Expr -> Expr '+' Term` but `Expr -> Term` (followed by `Term -> Factor` and then `Factor -> '(' Expr ')'`, in order to produce the pair of parentheses).
2. So, instead, the action at line 28 is matched. The update specified by this action is to proceed with updating the subtree produced from `Term`, so we move on to the second group.
3. Similarly, the actions at lines 33 and 38 match, and we are now updating the subtree $-1 + 2 * 3$ produced from `Expr` inside the parentheses. Note that, at this point, the parentheses have been preserved.
4. For this subtree, we should again try the first group of actions, and this time the first action (at line 26) matches, meaning that we should update the subtrees -1 and $2 * 3$ with `SUB (NUM 0) (NUM 4)` and `NUM 5` respectively.
5. For the update of -1, we go through the actions at lines 28, 33, 36, and 37, eventually updating the number 1 with 4, preserving the negation.
6. As for the update of $2 * 3$, all the actions in the group `Arith +> Term` fail, so we create a new concrete syntax tree from `NUM 5` by going through the actions at lines 33 and 37.

2.3.5 Parsing

So far we have been describing the putback semantics of the BiYacc program, but we may also work out its get semantics by intuitively reading the actions in Figure 1 from right to left (which might remind the reader of the usual Yacc actions from this opposite angle): The production rules for addition, subtraction, multiplication, and division expressions are converted to the corresponding constructors, and the production rule for negation expressions is converted to a subtraction whose left operand is zero. The other production rules are ignored and do not appear in the resulting abstract syntax tree.

3 Implementation

Although what a BiYacc program describes is an update, i.e., reflective printing, it can also be interpreted in the opposite direction, generating an abstract syntax tree from a concrete syntax tree, which is the more interesting part of the task of parsing. We realise the bidirectional interpretation by compiling BiYacc programs to BiFluX [16], a putback-based bidirectional programming language for XML document synchronisation. The actual semantics of parsing and reflective printing are not separately implemented, but derived from the underlying BiFluX program compiled from a BiYacc program. Although the actual semantics of parsing in terms of BiFluX is more complicated compared with the intuitive reading given in Section 2.3.5, it is derived for free, thanks to the fact that BiFluX is a bidirectional language. Inside the implementation, both well-formed strings and abstract syntax trees must be in XML format before they can be synchronised by BiFluX, so BiYacc is bundled with parsers and printers converting between well-formed strings/abstract syntax trees and XML, and the user can use BiYacc without knowing that synchronisation is done on XML documents internally.

4 Related work

Many domain-specific languages have been proposed for describing both a parser and a printer as a single program to remove redundancy and potential inconsistency between two separate descriptions of the parser and the pretty-printer. There are basically three approaches:

- The first approach is to extend the grammar description for parsers with information for printers and abstract syntax tree types [1, 5, 17, 18, 22]. For instance, Syn [1] is a language for writing extended BNF grammars with explicit annotations of layout for printing and implicit precedence (binding strength) by textual ordering; then, from a description in Syn, a parser and a printer can be automatically generated.

- The second approach is to extend the printer description with additional grammar information [12, 14]. For instance, FliPpr [14] is a program transformation system that uses program inversion to produce a context-free grammar parser from an enriched printer.

- The third approach is to provide a framework for describing both a parser and a printer hand in hand in a constructive way [19]; primitive parsers and their corresponding printers are first specified, and more complex ones are built up from simpler ones using a set of predefined constructors.

Different from our system, these approaches cannot (do not intend to) deal with synchronisation between the concrete and abstract syntax trees, in the sense that a printer will print the concrete syntax tree from scratch without taking into account the original concrete syntax tree if it already exists. In contrast, our method can not only do parsing and printing, but also enable flexible updates over the old concrete syntax tree when doing printing (i.e., update-based pretty-printing).

Various attempts have been made to build up update-based pretty-printers from (extended) parsers that can preserve comments, layouts, and structures in the original source text. One natural idea is to store all information that does not take part in the abstract syntax tree, which includes whitespace, comments and (redundant) parentheses, and the modified source code is reconstructed from the transformed abstract syntax tree by layout-aware pretty-printing [3, 13]. It becomes more efficient to take origin tracking as a mechanism to relate abstract terms with their corresponding concrete representation [4]. Origin tracking makes it possible to locate moved subtrees in the original text. All these systems are hard-coded in the sense that the user can neither control information to be preserved nor describe specific updating strategies to be used in printing.

Our work was greatly inspired by the recent progress on bidirectional transformations [2, 7, 9]. In particular, it is a nontrivial application of the new putback-based framework [6, 8, 15, 16] for bidirectional programming, where a single putback function has been proved to be powerful enough to fully control synchronisation behaviour.

Acknowledgements

This work was partially supported by the Project NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). It was also partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009, and the MOU grant of the National Institute of Informatics (NII), Japan. The authors would like to thank Jeremy Gibbons for discussions about the design of BiYacc, Tao Zan for helping with setting up the demo website, and the anonymous reviewers for their valuable comments and suggestions.

References

[1] R. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report Number 390, Computer Laboratory, University of Cambridge, 1966.

[2] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Model Transformation*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer-Verlag, 2009.

[3] M. de Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance*, pages 550–559. IEEE, 2002.

[4] M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In *International Conference on Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer-Verlag, 2012.

[5] J. Duregård and P. Jansson. Embedded parser generators. In *Haskell Symposium*, pages 107–117. ACM, 2011.

[6] S. Fischer, Z. Hu, and H. Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[8] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In *International Symposium on Formal Methods*, pages 1–15. Springer-Verlag, 2014.

[9] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl Seminar on Bidirectional Transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

[10] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *History of Programming Languages*, pages 1–55. ACM, 2007.

[11] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[12] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming Languages and Systems*, pages 273–287. Springer-Verlag, 1999.

[13] J. Kort and R. Lammel. Parse-tree annotations meet re-engineering concerns. In *International Workshop on Source Code Analysis and Manipulation*. IEEE, 2003.

[14] K. Matsuda and M. Wang. FliPpr: A prettier invertible printing system. In *European Conference on Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer-Verlag, 2013.

[15] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014.

[16] H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *Principles and Practice of Declarative Programming*, pages 147–158. ACM, 2014.

[17] A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

[18] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. Center for the Study of Language and Information/SRI, 2011.

[19] T. Rendel and K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Haskell Symposium*, pages 1–12. ACM, 2010.

[20] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[21] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[22] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

Trace-based Approach to Editability and Correspondence Analysis for Bidirectional Graph Transformations

Soichiro Hidaka
National Institute of Informatics, Japan
hidaka@nii.ac.jp

Martin Billes
Technical University of Darmstadt, Germany
martin.billes@cased.de

Quang Minh Tran
Daimler Center for IT Innovations, Technical University of Berlin, Germany
quang.tranminh@dcaiti.com

Kazutaka Matsuda
Tohoku University
kztk@ecei.tohoku.ac.jp

Abstract

Bidirectional graph transformation is expected to play an important role in model-driven software engineering where artifacts are often refined through compositions of model transformations, by propagating changes in the artifacts over transformations bidirectionally. However, it is often difficult to understand the correspondence among elements of the artifacts. The connections view elements have among each other and with source elements, which lead to restrictions of view editability, and parts of the transformation which are responsible for these relations, are not apparent to the user of a bidirectional transformation program.

These issues are critical for more complex transformations. In this paper, we propose an approach to analyzing the above correspondence as well as to classifying edges according to their editability on the target, in a compositional framework of bidirectional graph transformation where the target of a graph transformation can be the source of another graph transformation. These are achieved by augmenting the forward semantics of the transformations with explicit correspondence traces. By leveraging this approach, it is possible to solve the above issues, without executing the entire backward transformation.

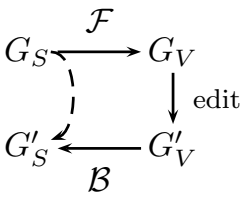
Keywords: Bidirectional Graph Transformation, Traceability, Editability

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

1 Introduction

Bidirectional transformations has been attracting interdisciplinary studies [5, 20]. In model-driven software engineering where artifacts are often refined through compositions of model transformations, bidirectional graph transformation is expected to play an important role, because it enables us to propagate changes in the artifacts over transformations bidirectionally.

A bidirectional transformation consists of forward ($\mathcal{F}$) and backward ($\mathcal{B}$) transformations [5, 20]. $\mathcal{F}$ takes a source model (here a source graph [15, 16]) G_S and transforms it into a view (target) graph G_V. $\mathcal{B}$ takes an updated view graph G'_V and returns an updated source graph G'_S, with possibly propagated updates.



In many, especially complex transformations, it is not immediately apparent whether a view edge has its origin in a particular source edge or a part (for example, operators or variables) in the transformation, and what that part is. Thus, it is not easy to tell where edits to the view edge are propagated back to. Moreover, in a setting in which usual strong well-behaved properties like PutGet [8] is relaxed to permit the backward transformation to be only partially defined, like WPutGet (WeakPutGet) [13, 16], it is not easy to predict whether a particular edit to the view succeeds. In particular, backward transformation rejects updates if (1) the label of the edited view edge appears as a constant of the transformation[1], (2) a group of view edges that are originated from the same source edge are edited inconsistently or (3) edits of view edges lead to changes in control flow (i.e., different branch is taken in the conditional expressions) in the transformation. If a lot of edits are made at once, it becomes increasingly difficult for the user to predict whether these edits are accepted by backward transformation. Bidirectional transformations are known for being hard to comprehend and predict [7]. Two features are desirable: 1) Explicit highlighting of correspondence between source, view and transformation 2) Classification of artifacts according to their editability. This way, prohibited edits leading to violation of predefined properties (well-behavedness) can be recognized by the user early.

Our bidirectional transformation framework called GRoundTram (Graph Roundtrip Transformation for Models) [18, 15, 16] features compositionality (e.g., the target of a sub-transformation can be the source of another sub-transformation), a user-friendly surface syntax, a tool for validating both models and transformations with respect to KM3 metamodels, and an performance optimization mechanism via transformation rewriting. It suffers from the above issues. To fix this, we have incorporated these features by augmenting the forward semantics with explicit trace information. Our main contribution is to externalize enough trace information through the augmentation and to utilize the information for correspondence and editability analysis. For the user, corresponding elements in the artifacts are highlighted with different colors according to editability and other properties.

There are some existing work with similar objectives. Traceability is studied enthusiastically in model-driven engineering [9, 25]. Van Amstel et al. [30] proposed a visualization of traces, but in the unidirectional model transformation setting. We focus on the backward transformation to give enough information on editability of the views for the programmer of the transformation and the users who edit the views. For classification of elements in the view, Matsuda and Wang's work [24], an extension of the semantic approach [31] to general bidirectionalization, is also capable of a similar classification, while we reserve opportunities to recommend a variety of consistent changes for more complex branching conditions. In addition, our approach can trace between nodes of the graph, not just edges. More details can be found in the related work section (Section 6).

The rest of the paper is organized as follows: Section 2 overviews our motivation and goal with a running example. More involved examples can be found on our project website at `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/`. Section 3 summarizes the semantics of our underlying graph data model, core graph language UnCAL [3], and its bidirectional interpretation [13]. Section 4 introduces an augmented semantics of UnCAL to generate trace information for the correspondence and editability analysis. Section 5 explains how the augmented forward semantics can be leveraged to realize correspondence and editability analysis. An implementation is found at the above website. Section 6 discusses related work, and Section 7 concludes the paper with future work. The long version of our paper [12] includes some non-essential technical details that are omitted in this paper.

[1] In this case, the constant in the transformation is pointed out so that user can consider changing it in the transformation directly.

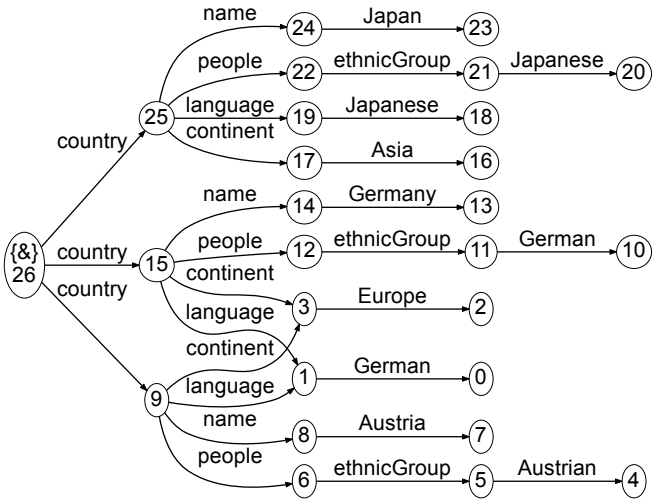


Figure 1: Example Source Graph

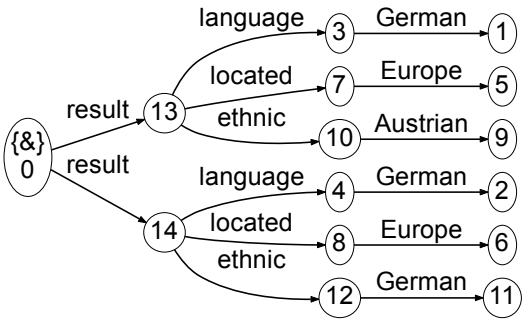


Figure 2: View Graph Generated by Transformation of the Graph in Fig. 1

2 Motivating Example

This section exemplifies the importance of the desirable features mentioned in Sect. 1. Let us consider the source graph in Fig. 1 consisting of fact book information about different countries, and suppose we want to extract the information for European countries as the view graph in Fig. 2. This transformation can be written in UnQL (the surface language as a syntactic sugar of the target language UnCAL) as below.

```
select {result: {ethnic: $e, language: $lang, located: $cont}}
where {country: {name:$g, people: {ethnicGroup: $e},
                 language: $lang, continent: $cont}} in $db,
      {$l:$Any} in $cont, $l = Europe
```

Listing 1: Transformation in UnQL

S1: In the view graph (Fig. 2), three edges have identical labels "German" $(3, \text{German}, 1)$, $(4, \text{German}, 2)$ and $(12, \text{German}, 11)$, but have different origins in the source graph and are produced by different parts in the transformation. For example, the edge $\zeta = (3, \text{German}, 1)$ is the language of Germany and is a copy of the edge $(1, \text{German}, 0)$ of the source graph (Fig. 1). On the other hand, ζ has nothing to do with the edge $(11, \text{German}, 10)$ of the source graph despite identical labels. This later edge denotes the ethnic group instead. In addition, ζ is copied by the graph variable $\$lang$ in the **select** part of the transformation. Other graph variables and edge constructors do not participate in creating ζ. It would be much easier for the user to understand, if the system visually highlights corresponding elements between source graph, view graph and transformation to increase comprehensibility.

S2: In this example, the non-leaf edges of the view graph ("result", "located", "language" and "ethnic") are constant edges in the sense that they cannot be modified by the user in the view. Ideally, the system should make such constant edges easily recognizable to prevent the edits in the view and guide the user to make an edit to the constant in the transformation instead.

S3: In another scenario, the user decides that the language of Germany should better be called "German (Germany)" and the language of Austria be called "Austrian German" and thus rename the view edges $(3, \text{German}, 1)$ and $(4, \text{German}, 2)$ to $(3, \text{German (Germany)}, 1)$ and $(4, \text{Austrian German}, 2)$ accordingly. However, the backward transformation rejects this modification because these two view edges originate from the language "German" in a single edge of the source graph. The backward propagation of two edits would conflict at the source edge. Ideally, the system would highlight groups of edges that could cause the conflict, and would prohibit triggering the backward transformation in that case.

S4: Finally, suppose both of the edges labeled "Europe" in the view graph are edited to "Eurasia". Although this time these updates would be propagated to the same originating source edge $(3, \text{Europe}, 2)$ without conflict, since the transformation depends on the label of this edge, changing it would lead to the selection of another conditional path in the transformation in a subsequent forward transformation. The renaming would cause an empty view after a round-trip of backward and forward transformation. To prevent this, such edits are rejected

in our system. Changes in the control flow are very difficult or impossible to predict if the transformation is too complex. We can assist the prediction by highlighting the conditional branches involved.

Formal property As a remark, our editability checking is sound in the sense that the edits that passed the checking always succeed. We provide a proof sketch in Section 5.

Using this property and analysis, additional interesting observations can be made. In the Class2RDB example on our project website, there is no edge in the view for which the above last warning is caused. Thus, the transformation is "fully polymorphic" in the sense that all edges that come from source graph are free from label inspection in the transformation. Note that this editability property may not be fully determined by just looking at the transformation. Non-polymorphic transformations may still produce views that avoid the warning. For example,

select $\$g$ **where** {a:$\$g$} **in** $\$db$

is not polymorphic, but the edits on edges in the view never cause condition expressions to be changed, because they are not inspected (they are inspected by bulk semantics explained in the next section but left unreachable).

Tracing mechanism Though the editability analysis is powerful enough to, for example, cope with complex interactions between graph variables and label variables, the underlying idea is simple. For the analysis of a chain of transformations, if an edge ζ is related to ζ' in the first transformation and ζ' is related to ζ'' in the second transformation, then trace information allows to relate ζ and ζ'', possibly with the occurrence information of the language constructs in the transformation that created these edges. The variable binding environment is extended to cope with this analysis.

Support of editing operations other than edge renaming As another remark, this paper focuses on updates on edge labels, and trace information is designed as such. However, we can relatively easily extend the data structure of the trace to cope with edge deletion. The data structure already supports node tracing, so insertion operation can be supported by highlighting the node in the source on which a graph will be inserted.

Support of permissive bidirectional transformation The bidirectional transformation we are dealing with under relaxed notion of consistency, i.e., WeakPutGet as mentioned in the introduction, in fact is permissive in users edits. For example, in the above scenario **S3**, if the user edits only one of the view edge, then the backward transformation successfully propagates the changes to the source edge, though the updated view graph on the whole is not perfectly consistent if we define source s and view v to be consistent if and only if $gets = v$ where *get* is the forward transformation, because another forward transformation from the updated source would propagate the new edge label to the other copy edge in the view graph. The perfectly consistent view graph would have been the one which all the edges originating from a common source edge are updated to the same value. However, it might be hard for the users to identify all these copies. Therefore, our system accepts updates in which only one copy is updated in the view. Borrowing the notion of degree of consistency by Stevens [28], such updated views are (strictly) less consistent than the perfect one but still tolerated.

3 Preliminaries

We use the UnCAL (Unstructured CALculus) query language [3]. UnCAL has an SQL-like syntactic sugar called UnQL (Unstructured Query Language) [3]. Listing 1 is written in UnQL. Bidirectional execution of graph transformation in UnQL is achieved by desugaring the transformation into UnCAL and then bidirectionally interpreting it [13]. This section explains the graph data model we use, as well as the UnCAL and UnQL languages.

3.1 UnCAL Graph Data Model

UnCAL graphs are multi-rooted and edge-labeled with all information stored in edge labels ranging over $Label \cup \{\varepsilon\}$ ($Label_\varepsilon$), node labels are only used as identifiers. There is no order between outgoing edges of a node. The notion of graph equivalence is defined by bisimulation; so equivalence between the graphs is efficiently determined [3], and graphs can be normalized [16] up to isomorphism.

Fig. 3 shows examples of our graphs. We represent a graph by a quadruple (V, E, I, O). V is the set of nodes, E the set of edges ranging over the set $Edge_\varepsilon$, where an edge is represented by a triple of source node, label and destination node. $I : Marker \to V$ is a function that identifies the roots (called *input nodes*) of a graph. Here, *Marker* is the set of markers of which element is denoted by $\&x$. We may call a marker in $\mathrm{dom}(I)$ (domain of I) an *input marker*. A special marker $\&$ is called the default marker. $O \subseteq V \times Marker$ assigns nodes with markers called *output markers*. If $(v, \&m) \in O$, v is called an *output node*. Intuitively output nodes serve as "exit points" where

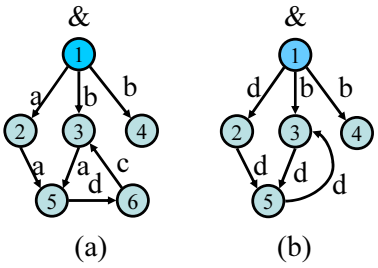


Figure 3: Cyclic graph examples

$$
\begin{array}{rll}
e ::= & \{\} \mid \{l : e\} \mid e \cup e \mid \texttt{\&}x := e \mid \texttt{\&}y \mid () & \\
\mid & e \oplus e \mid e \mathbin{@} e \mid \textbf{cycle}(e) & \{ \text{ constructor } \} \\
\mid & \$g & \{ \text{ graph variable } \} \\
\mid & \textbf{if } l = l \textbf{ then } e \textbf{ else } e & \{ \text{ conditional } \} \\
\mid & \textbf{let } \$g = e \textbf{ in } e \mid \textbf{llet } \$l = l \textbf{ in } e & \{ \text{ variable binding } \} \\
\mid & \textbf{rec}(\lambda(\$l, \$g).e)(e) & \{ \text{ structural recursion application } \} \\
l ::= & a \mid \$l & \{ \text{ label } (a \in \mathit{Label}) \text{ and label variable } \}
\end{array}
$$

Figure 4: Core UnCAL Language

input nodes serve as "entry points". For example, the graph in Fig. 3 (a) is represented by (V, E, I, O), where $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, \mathtt{a}, 2), (1, \mathtt{b}, 3), (1, \mathtt{b}, 4), (2, \mathtt{a}, 5), (3, \mathtt{a}, 5), (5, \mathtt{d}, 6), (6, \mathtt{c}, 3)\}$, $I = \{\texttt{\&} \mapsto 1\}$, and $O = \{\}$. Each component of the quadruple is denoted by the "." syntax, such as $g.\mathrm{V}$ for V of graph $g = (V, E, I, O)$.

The type of a graphs is defined as the pair of the set of its input markers $\mathcal{X}$ and the set of output markers $\mathcal{Y}$, denoted by $DB^{\mathcal{X}}_{\mathcal{Y}}$. The graph in Fig. 3 (a) has type $DB^{\{\texttt{\&}\}}_{\emptyset}$. The superscript may be omitted, if the set is $\{\texttt{\&}\}$, and the subscript likewise, if the set is empty. The type of this graph is simply denoted by DB.

3.2 UnCAL Query Language

Graphs can be created in the UnCAL query language [3], where there are nine graph constructors (Fig. 4) whose semantics is illustrated in Fig. 5. We use hooked arrows ($\hookrightarrow$) stacked with the constructor to denote the computation by the constructors where the left-hand side is the operand(s) and the right-hand side is the result.

There are three nullary constructors. () constructs a graph without any nodes or edges, so $\mathcal{F}[\![()]\!] \in DB^{\emptyset}$, where $\mathcal{F}[\![e]\!]$ denotes the (forward) evaluation of expression e. The constructor {} constructs a graph with a node with default input marker (&) and no edges, so $\mathcal{F}[\![\{\}]\!] \in DB$. &$y$ constructs a graph similar to {} with additional output marker &y associated with the node, i.e., $\mathcal{F}[\![\texttt{\&}y]\!] \in DB_{\{\texttt{\&}y\}}$.

The edge constructor $\{\_ : \_\}$ takes a label l and a graph $g \in DB_{\mathcal{Y}}$, constructs a new root with the default input marker with an edge labeled l from the new root to $g.\mathrm{I}(\texttt{\&})$; thus $\{l : g\} \in DB_{\mathcal{Y}}$. The union $g_1 \cup g_2$ of

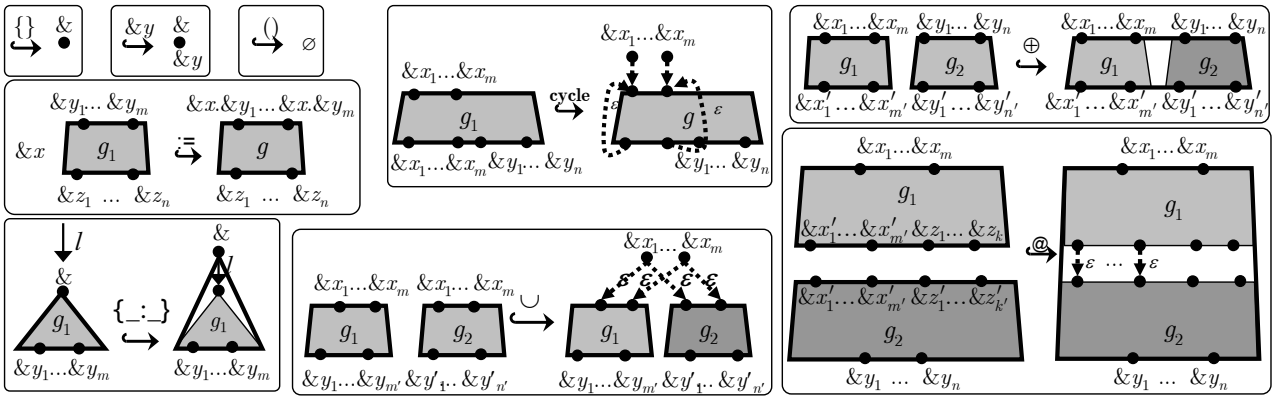


Figure 5: Graph Constructors of UnCAL

graphs $g_1 \in DB^{\mathcal{X}}_{\mathcal{Y}_1}$ and $g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}_2}$ with the identical set of input markers $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, constructs m new input nodes for each $\&x_i \in \mathcal{X}$, where each node has two ε-edges to $g_1.\mathrm{I}(\&x_i)$ and $g_2.\mathrm{I}(\&x_i)$. Here, ε-edges are similar to ε-transitions in automata and used to connect components during the graph construction. Clearly, $g_1 \cup g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}$.

The input node renaming operator := takes a marker $\&x$ and a graph $g \in DB^{\mathcal{Y}}_{\mathcal{Z}}$ with $\mathcal{Y} = \{\&y_1, \ldots, \&y_m\}$, and returns a graph whose input markers are prepended by $\&x$, thus $(\&x := g) \in DB^{\&x.\mathcal{Y}}_{\mathcal{Z}}$ where the dot "." concatenates markers and forms a monoid with & , i.e., $\&.\&x = \&x.\& = \&x$ for any marker $\&x \in \mathit{Marker}$, and $\&x.\mathcal{Y} = \{\&x.\&y_1, \ldots, \&x.\&y_m\}$ for $\mathcal{Y} = \{\&y_1, \ldots, \&y_m\}$. In particular, when $\mathcal{Y} = \{\&\}$, the := operator just assigns a new name to the root of the operand, i.e., $(\&x := g) \in DB^{\{\&x\}}_{\mathcal{Y}}$ for $g \in DB_{\mathcal{Y}}$.

The disjoint union $g_1 \oplus g_2$ of two graphs $g_1 \in DB^{\mathcal{X}}_{\mathcal{X}'}$ and $g_2 \in DB^{\mathcal{Y}}_{\mathcal{Y}'}$ with $\mathcal{X} \cap \mathcal{Y} = \emptyset$, the resultant graph inherits all the markers, edges and nodes from the operands, thus $g_1 \oplus g_2 \in DB^{\mathcal{X} \cup \mathcal{Y}}_{\mathcal{X}' \cup \mathcal{Y}'}$.

The remaining two constructors connect output and input nodes with matching markers by ε-edges. $g_1 \mathbin{@} g_2$ appends $g_1 \in DB^{\mathcal{X}}_{\mathcal{X}' \cup \mathcal{Z}}$ and $g_2 \in DB^{\mathcal{X}' \cup \mathcal{Z}'}_{\mathcal{Y}}$ by connecting the output and input nodes with a matching subset of markers $\mathcal{X}'$, and discards the rest of the markers, thus $g_1 \mathbin{@} g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}}$. An idiom $\&x' @ g_2$ projects (selects) one input marker $\&x'$ and renames it to default ($\&$), while discarding the rest of the input markers (making them unreachable). The cycle construction $\mathbf{cycle}(g)$ for $g \in DB^{\mathcal{X}}_{\mathcal{X} \cup \mathcal{Y}}$ with $\mathcal{X} \cap \mathcal{Y} = \emptyset$ works similarly to @ but in an intra-graph instead of inter-graph manner, by connecting output and input nodes of g with matching markers $\mathcal{X}$, and constructs copies of input nodes of g, each connected with the original input node by an ε-edge. The output markers in $\mathcal{Y}$ are left as is.

It is worth noting that any graph in the data model can be expressed by using these UnCAL constructors (up to bisimilarity), where the notion of bisimilarity is extended to ε-edges [3].

The semantics of conditionals is standard, but the condition is restricted to label equivalence comparison. There are two kinds of variables: label variables and graph variables. Label variables, denoted $\$l, \$l_1$ etc., bind labels while graph variables denoted $\$g, \$g_1$ etc., bind graphs. They are introduced by structural recursion operator **rec**, whose semantics is explained below by example. The variable binders **let** and **llet** having standard meanings are our extensions used for optimization by rewriting [14].

We take a look at the following concrete transformation in UnCAL that replaces every label `a` by `d` and contracts edges labeled `c`.

$$\begin{aligned}\mathbf{rec}(\lambda(\$l, \$g).\ &\mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{d} : \&^1\}^2 \\ &\mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ \{\varepsilon : \&^3\}^4 \\ &\mathbf{else}\qquad\qquad\ \{\$l : \&^5\}^6)(\$db)^7\end{aligned}$$

If the graph variable $\$db$ is bound to the graph in Fig. 3 (a), the result of the transformation will be the one in Fig. 3 (b). We call the first operand of **rec** the *body* expression and the second operand the *argument* expression. In the above transformation, the body is an **if** conditional, while the argument is the variable reference $\$db$. We use $\$db$ as a special global variable to represent the input of the graph transformation. For the sake of bidirectional evaluation (and also used in our tracing in this paper), we superscribe UnCAL expressions with their code position $p \in \mathit{Pos}$ where Pos is the set of position numbers. For instance, in the example above, the numbers 1 and 2 in $\{\mathtt{d} : \&^1\}^2$ denote the code positions of the graph constructors & and $\{\mathtt{d} : \&\}$, respectively.

Fig. 6 shows the *bulk* semantics of **rec** for the example. It is "bulk" because the body of **rec** can be evaluated in parallel for each edge and the subgraph reachable from the target node of the edge (which are correspondingly bound to variables $\$l$ and $\$g$ in the body).

In the bulk semantics, the node identifier carries some information which has the following structure [13] *StrID*:

$$\begin{aligned}\mathit{StrID} ::= \ &\mathit{SrcID} \\ \mid\ &\mathsf{Code}\ \mathit{Pos}\ \mathit{Marker} \\ \mid\ &\mathsf{RecN}\ \mathit{Pos}\ \mathit{StrID}\ \mathit{Marker} \\ \mid\ &\mathsf{RecE}\ \mathit{Pos}\ \mathit{StrID}\ \mathit{Edge},\end{aligned}$$

where the base case (*SrcID*) represents the node identifier in the input graph, Code p $\&x$ denotes the nodes constructed by {}, $\{\_ : \_\}$, $\&y$, $\cup$ and **cycle** where $\&x$ is the marker of the corresponding input node of the operand(s) of the constructor. Except for $\cup$, the marker is always default and thus omitted. RecN p v $\&z$ denotes the node created by **rec** at position p for node v of the graph resulting from evaluating the argument expression. For example, in Fig. 6, the node RN 7 1, originating from node 1, is created by **rec** at position 7

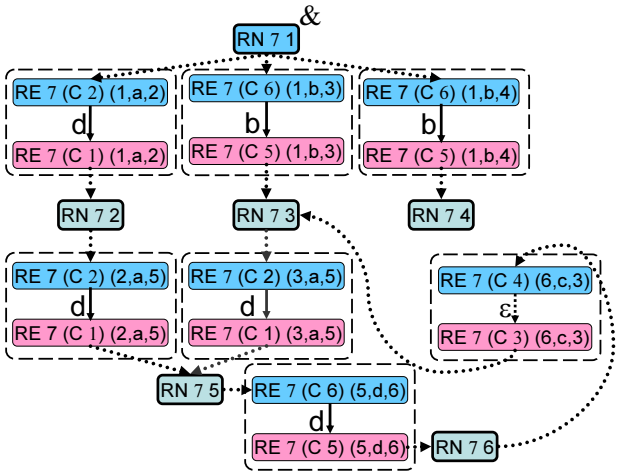


Figure 6: Bulk Semantics by Example

(RecN is abbreviated to RN in the figure for simplicity, and similarly Code to C and RecE to RE). We have six such nodes, one for each in the input graph. Then we evaluate the body expression for each binding of $\$l$ and $\$g$. For the edge $(1, \mathtt{a}, 2)$, the result will be $(\{(\mathsf{C}\ 2), (\mathsf{C}\ 1)\}, \{(\mathsf{C}\ 2, \mathtt{d}, \mathsf{C}\ 1)\}, \{\mathtt{\&} \mapsto \mathsf{C}\ 2\}, \{(\mathsf{C}\ 2, \mathtt{\&})\})$, with the nodes C 2 and C 1 constructed by $\{\_ : \_\}$ and &, respectively. For the shortcut edges, an ε-edge is generated similarly. Then each node v of such results for edge ζ is wrapped with the trace information RE like RE p v ζ for **rec** at position p. These results are surrounded by round squares drawn with dashed lines in Fig. 6. They are then connected together according to the original shape of the graph as depicted in Fig. 6. For example, the input node $\boxed{\mathsf{RE}\ 7\ (\mathsf{C}\ 2)\ (1, \mathtt{a}, 2)}$ is connected with $\boxed{\mathsf{RN}\ 7\ 1}$. After removing the ε-edges and flattening the node IDs, we obtain the result graph in Fig. 3 (b).

Our bidirectional transformation in UnCAL is based on its bidirectional evaluation, whose semantics is given by $\mathcal{F}[\![\_]\!]$ and $\mathcal{B}[\![\_]\!]$ as follows. $\mathcal{F}[\![e]\!]\rho = G$ is the forward semantics applied to UnCAL query e with source variable environment ρ, which includes a global variable binding $\$db$ the input graph. $\mathcal{B}[\![e]\!](\rho, G') = \rho'$ produces the updated source ρ' given the updated view graph G' and the original source ρ.

Bidirectional transformations need to satisfy round-trip properties [5, 27], while ours satisfy the GetPut and WPutGet properties [13], which are:

$$\frac{\mathcal{F}[\![e]\!]\rho = G_V}{\mathcal{B}[\![e]\!](\rho, G_V) = \rho}\ \text{(GetPut)} \qquad \frac{\mathcal{B}[\![e]\!](\rho, G'_V) = \rho' \quad \mathcal{F}[\![e]\!]\rho' = G''_V}{\mathcal{B}[\![e]\!](\rho, G''_V) = \rho'}\ \text{(WPutGet)}$$

where GetPut says that when the view is not updated after forward transformation, the result of the following backward transformation agrees with the original source, and W(Weak)PutGet (a.k.a. *weak invertibility* [6], a weaker notion of PutGet [8] or Correctness [27] or Consistency [2] because of the rather arbitrary variable reference allowed in our language) demands that for a second view graph G''_V which is returned by $\mathcal{F}[\![e]\!]\rho'$, that backward transformation $\mathcal{B}[\![e]\!](\rho, G''_V)$ using this second view graph as well as the original source environment ρ (from the first round of forward transformation) returns ρ' again unchanged.

In the backward evaluation of **rec**, the final ε-elimination to hide them from the user is reversed to restore the shape of Fig. 6, and then the graph is decomposed with the help of the structured IDs, and then the decomposed graph is used for the backward evaluation of each body expression. The backward evaluation produces the updated variable bindings (in this body expression we get the bindings for $\$l$, $\$g$ and $\$db$ and merge them to get the final binding of $\$db$). For example, the update of the edge label of $(1, \mathtt{b}, 3)$ in the view to x is propagated via the backward evaluation of the body $\{\$l : \mathtt{\&}\}$, which produces the binding of $\$l$ updated with x and is reflected to the source graph with edge $(1, \mathtt{b}, 3)$, replaced by $(1, \mathtt{x}, 3)$.

UnQL as a Textual Surface Syntax of Bidirectional Graph Transformation

We use the surface language UnQL [3] for bidirectional graph transformation. An UnQL expression can be translated into UnCAL, a process referred to as desugaring. We highlight the essential part of the translation in the following. Please refer to [3, 19, 17] for details. The expression (directly after the **select** clause) appears

in the innermost body of the nested **rec** in the translated UnCAL. The edge constructor expression is directly passed through, while the graph variable pattern in the **where** clause and corresponding references are translated into combinations of graph variable bindings in nested **rec**s as well as references to them in the body of **rec**s. The following example translates an UnQL expression into an equivalent UnCAL one.

$$
\begin{array}{l}
\textbf{select } \{\text{res:}\$db\} \\
\textbf{where } \{\text{a:}\$g\} \textbf{ in } \$db, \\
\qquad \{\text{b:}\$g\} \textbf{ in } \$db
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\textbf{rec}(\lambda(\$l,\$g).\ \textbf{if } \$l = \text{a} \\
\quad \textbf{then rec}(\lambda(\$l',\$g).\ \textbf{if } \$l' = \text{b} \\
\qquad \textbf{then } \{\text{res:}\$db\} \\
\qquad \textbf{else } \{\})(\$db) \\
\quad \textbf{else } \{\})(\$db).
\end{array}
$$

4 Trace-augmented Forward Semantics of UnCAL

This section describes the forward semantics of UnCAL augmented with explicit correspondence traces. In the trace, every view edge and node is mapped to a corresponding source edge or node or a part of the transformation. The path taken in the transformation is also recorded in the trace for the view elements.

The trace information is utilized for (1) correspondence analysis, where a selected source element is contrasted with its corresponding view element(s) by highlighting them, and likewise, a selected view element is contrasted with its corresponding parts in source and transformation, and for (2) editability analysis, classifying edges by origins to (2-1) pre-reject the editing of edges that map to the transformation, (2-2) pre-reject the conflicting editing of view edges whose edit would be propagated to the same source edge, (2-3) warn the edit that could violate WPutGet by changing branching behavior of **if** by highlighting the branch conditions that are affected by the edit.

The augmented forward evaluation $\mathcal{F}[\![\_]\!] : \mathit{Expr} \to \mathit{Env} \to \mathit{Graph} \times \mathit{Trace}$ takes an UnCAL expression and an environment as arguments, and produces a target graph and trace. The shaded part represents the augmented part and we apply the same shading in the following. The trace maps an edge $\in \mathit{Edge}$ (resp. a node $\in \mathit{Node}$) to a source edge (resp. source node) or a code position, preceded by zero or more code positions that represent the corresponding language constructs involved in the transformation that produced the edge (resp. the node). The preceding parts are used for the warning in (2-3) above. Thus

$$
\begin{array}{rcl}
\mathit{Trace} & = & \mathit{Edge} \cup \mathit{Node} \to \mathit{Trace}_{\mathrm{E}} \cup \mathit{Trace}_{\mathrm{V}} \\
\mathit{Trace}_{\mathrm{E}} & ::= & \mathit{Pos} : \mathit{Trace}_{\mathrm{E}} \mid [\mathit{Edge} \mid \mathit{Pos}] \\
\mathit{Trace}_{\mathrm{V}} & ::= & \mathit{Pos} : \mathit{Trace}_{\mathrm{V}} \mid [\mathit{Node} \mid \mathit{Pos}]
\end{array}
$$

The environment *Env* represents bindings of graph variables and label variables. Each graph variable is mapped to a graph with a trace, while each label variable is mapped to a label with a trace that contains only edge mapping. Thus

$$\mathit{Env} = \mathit{Var} \to (\mathit{Graph} \times \mathit{Trace}) \cup (\mathit{Label} \times \mathit{Trace}_{\mathrm{E}}).$$

Given source graph g_s, the top level environment ρ_0 is initialized as follows.

$$\rho_0 = \{\$db \mapsto (g_s, \{\zeta \mapsto [\zeta] \mid \zeta \in g_s.\mathrm{E}\} \cup \{v \mapsto [v] \mid v \in g_s.\mathrm{V}\})\} \qquad (\textsc{InitEnv})$$

As idioms used in the following, we introduce two auxiliary functions of type $\mathit{Trace} \to \mathit{Trace}$: prep_p to prepend code position $p \in \mathit{Pos}$ to traces, and $\mathsf{rece}_{p,\zeta}$ to "wrap" the nodes in the domain of traces with RecE constructor to adjust to bulk semantics.

$$
\begin{array}{l}
\mathsf{prep}_p\, t = \{\quad x \quad \mapsto p{:}\tau \mid (x \mapsto \tau) \in t\} \\
\mathsf{rece}_{p,\zeta}\, t = \{(f\ x) \mapsto \quad \tau \mid (x \mapsto \tau) \in t\} \\
\qquad \textbf{where } fx = \begin{cases} (\mathsf{RecE}\ p\ u\ \zeta, l, \mathsf{RecE}\ p\ v\ \zeta) & \text{if } x = (u,l,v) \in \mathit{Edge} \\ \mathsf{RecE}\ p\ x\ \zeta & \text{if } x \in \mathit{Node} \end{cases}
\end{array}
$$

Now we describe the semantics $\mathcal{F}[\![\_]\!]$. The graph component returned by the semantics is the same as that of [13] and recapped in Section 3, so we focus here on the trace parts. The subscripts on the left of the constructor expressions represent the result graph of the constructions. For the constructors, we only show the semantics of

some representative ones. See the long version [12] for the rest.

$$\mathcal{F}[\![\{\}^p]\!]_\rho = ({}_G\{\}^p, \{G.\mathrm{I}(\&) \mapsto [p]\}) \tag{T-EMP}$$

$$\mathcal{F}[\![e_1 \cup^p e_2]\!]_\rho = ({}_G(g_1 \cup^p g_2), (t_1 \cup t_2 \cup \{v \mapsto [p] \mid (\&x \mapsto v) \in G.\mathrm{I}\})) \tag{UNI}$$
$$\textbf{where} \quad ((g_1, t_1), (g_2, t_2)) = (\mathcal{F}[\![e_1]\!]\rho, \mathcal{F}[\![e_2]\!]\rho)$$

$$\mathcal{F}[\![\{e_\mathrm{L} : e\}^p]\!]_\rho = ({}_G\{l : g\}^p, \{(G.\mathrm{I}(\&), l, g.\mathrm{I}(\&)) \mapsto \tau, G.\mathrm{I}(\&) \mapsto [p]\} \cup t) \tag{Edg}$$
$$\textbf{where} \quad ((l, \tau), (g, t)) = (\mathcal{F}_\mathrm{L}[\![e_\mathrm{L}]\!]_\rho, \mathcal{F}[\![e]\!]_\rho)$$

For the constructor $\{\}$, the trace maps the only node ($G.\mathrm{I}(\&)$) created, to the code position p of the constructor (T-EMP). The binary graph constructors $\cup$, $\oplus$ and @ returns the traces of both subexpressions, in addition to the trace created by themselves. The graph union's trace maps newly created input nodes to the code position (UNI). For the edge-constructor (Edg), the correspondence between the created edge and its trace created by the label expression e_L is established, while the newly created node is mapped to the code position of the constructor.

For label expression evaluation $\mathcal{F}_\mathrm{L}[\![\_]\!] : \mathit{Expr}_\mathrm{L} \to \mathit{Env} \to \mathit{Label} \times \mathit{Trace}_\mathrm{E}$, the trace that associates the label to the corresponding edge or code position is accompanied with the resultant label value.

$$\mathcal{F}_\mathrm{L}[\![\mathsf{a}^p]\!]_\rho = (\mathsf{a}, [p]) \tag{LCNST}$$

$$\mathcal{F}_\mathrm{L}[\![\$l^p]\!]_\rho = (l, p : \tau) \tag{LVAR}$$
$$\textbf{where } (l, \tau) = \rho(\$l)$$

Label literal expressions (LCNST) record their code positions, while label variable reference expressions (LVAR) add their code positions to the traces that are registered in the environment.

The label variable binding expression (LLET) registers the trace to the environment and passes it to the forward evaluation of the body expression e. Graph variable binding expression (LET) is treated similarly, except it handles graphs and their traces. Graph variable reference (VAR) retrieves traces from the environment and add the code position of the variable reference to it.

$$\mathcal{F}[\![\textbf{llet}^p\ \$l = e_\mathrm{L}\ \textbf{in}\ e]\!]_\rho = \mathcal{F}[\![e]\!]_{\rho \cup \{\$l \mapsto (l, p:\tau)\}} \tag{LLET}$$
$$\textbf{where } (l, \tau) = \mathcal{F}_\mathrm{L}[\![e_\mathrm{L}]\!]_\rho$$

$$\mathcal{F}[\![\textbf{let}^p\ \$g = e_1\ \textbf{in}\ e_2]\!]_\rho = \mathcal{F}[\![e_2]\!]_{\rho \cup \{\$g \mapsto (g, \mathsf{prep}_p\, t)\}} \tag{LET}$$
$$\textbf{where } (g, t) = \mathcal{F}[\![e_1]\!]_\rho$$

$$\mathcal{F}[\![\$g^p]\!]_\rho = (g, \mathsf{prep}_p\, t) \tag{VAR}$$
$$\textbf{where } (g, t) = \rho(\$g)$$

$$\mathcal{F}\left[\!\!\left[\begin{array}{rl}\textbf{if}^p(e_\mathrm{L} = e'_\mathrm{L})\ \textbf{then}\ e_\mathrm{true} \\ \textbf{else}\ \ e_\mathrm{false}\end{array}\right]\!\!\right]_\rho = (g, \mathsf{prep}_p\, t) \tag{IF}$$
$$\textbf{where } ((l, \_), (l', \_)) = (\mathcal{F}_\mathrm{L}[\![e_\mathrm{L}]\!], \mathcal{F}_\mathrm{L}[\![e'_\mathrm{L}]\!])$$
$$b = (l = l')$$
$$(g, t) = \mathcal{F}[\![e_b]\!]_\rho$$

Structural recursion **rec** (REC) introduces a new environment for the label ($\$l$) and graph ($\$g$) variable that includes traces inherited from the traces generated by the argument expression e_a, augmented with the code position p of **rec**. g_v denotes the subgraph of graph g that is reachable from node v. t_v denotes a trace that are restricted to the subgraph reachable from node v. The function $M : \mathit{Edge} \to (\mathit{Graph} \times \mathit{Trace})$ takes an edge and returns the pair of the graph created by the body expression e_b for the edge, and the trace associated with the graph. t_ζ is the trace generated by adjusting the trace returned by M with the node structure introduced by **rec**. $\mathsf{compose}^p_{\mathbf{rec}}$ is the bulk semantics explained in Sect. 3.2 using Fig. 6, for the input graph g and the input/output marker $\mathcal{Z}$ of e_b, where V_{RecN} denotes the nodes with structured ID RecN.

$$\mathcal{F}[\![\mathbf{rec}^p_{\mathcal{Z}}(\lambda(\$l,\$g).e_{\mathrm{b}})(e_{\mathrm{a}})]\!]_\rho = (g', \bigcup_{\zeta \in g.\mathrm{E}} t_\zeta \cup t'_{\mathrm{V}}) \qquad \text{(Rec)}$$

$$\begin{aligned}
\textbf{where}\ (g,t) &= \mathcal{F}[\![e_{\mathrm{a}}]\!]\rho \\
M &= \{\zeta \mapsto \mathcal{F}[\![e_{\mathrm{b}}]\!]_{\rho'} \mid \zeta \in g.\mathrm{E}, \zeta \neq \varepsilon, (u,l,v) = \zeta, \\
&\qquad \rho' = \rho \cup \{\$l \mapsto (l, p : t(\zeta)), \$g \mapsto (g_v, \mathsf{prep}_p\, t_v)\}\} \\
g' &= (V_{\mathsf{RecN}} \cup \ldots, _, _, _) = \mathsf{compose}^p_{\mathbf{rec}}(M, g, \mathcal{Z}) \\
t'_{\mathrm{V}} &= \{v \mapsto [p] \mid v \in V_{\mathsf{RecN}}\} \\
t_\zeta &= \mathsf{rece}_{p,\zeta}(\mathsf{prep}_p\, \pi_2(M(\zeta)))
\end{aligned}$$

The above semantics collects all the necessary trace information whose utilization is described in the next section. Even though the tracing mechanisms are defined for UnCAL, they also work straightforwardly for UnQL, based on the observation that when an UnQL query is translated into UnCAL, all edge constructors and graph variables in the UnQL query creating edges in the view graph are preserved in the UnCAL query. One limitation is: in our system, the bidirectional interpreter of UnCAL optionally rewrites expressions for efficiency. However, due to reorganization of expressions during the rewriting, we currently support neither tracing UnCAL nor tracing UnQL if the rewriting is activated.

5 Correspondence and Editability Analysis

This section elaborates utilization of the traces defined in Sect. 4 for the correspondence and editability analysis motivated in Sect. 2. Soundness of this analysis is discussed at the end of this section.

Given transformation e, environment ρ_0 (defined by InitEnv), and the corresponding trace t for $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ through semantics given in Sect. 4, the trace (represented as a list) for view edge ζ has the following form

$$t(\zeta) = p_1 : p_2 : \ldots : p_n : [x] \quad (n \geq 0)$$

where x is the origin, and $x = \zeta' \in \mathit{Edge}$ if ζ is a copy of ζ' in the source graph, or $x = p \in \mathit{Pos}$ if the label of ζ is a copy of the label constant at position p in the transformation. $p_1, p_2, \ldots, p_n$ represent code positions of variable definitions/references and conditionals that conveyed ζ.

For view graph g and trace t, define the function $\mathsf{origin} : \mathit{Edge} \to \mathit{Edge} \cup \mathit{Pos}$ and its inverse:

$$\begin{aligned}
\mathsf{origin}\ \ \zeta &= \mathsf{last}(t(\zeta)) \\
\mathsf{origin}^{-1}\ x &= \{\zeta \mid \zeta \in g.\mathrm{E}, \mathsf{origin}\, \zeta = x\}
\end{aligned}$$

Correspondence is then the relation between the domain and image of trace t, and various individual correspondence can be derived, the most generic one being $R : \mathit{Edge} \cup \mathit{Node} \cup \mathit{Pos} \times \mathit{Edge} \cup \mathit{Node} = \{(x', x) \mid (x \mapsto \tau) \in t, x' \in \tau\}$, meaning that x' and x is related if $(x', x) \in R$. Source-target correspondence being $\{(x', x) \mid (x \mapsto \tau) \in t, x' = \mathsf{last}\, \tau, x' \in (\mathit{Node} \cup \mathit{Edge})\}$. Using origin and origin^{-1}, corresponding source, transformation and view elements can be identified in both directions. When a view element such as the edge $\zeta = (4, \mathrm{German}, 2)$ in Fig. 2 is given, we can find the corresponding source edge $\mathsf{origin}(\zeta) = (1, \mathrm{German}, 0)$, which will be updated if we change ζ. In contrast, given the view edge $(14, \texttt{language}, 4)$, the code position of the label constant `lang` in $\{\texttt{lang} : \$e\}$ of the **select** part in Listing 1 is obtained. Given the view edge $\zeta = (3, \texttt{German}, 1)$, the code positions of the graph variables $\$\mathit{lang}$ of the **select** part and $\$\mathit{db}$ in Listing 1 are obtained, utilizing code positions in $p_1, \ldots, p_n$, because $t\ \zeta$ includes such positions. These graph variables copy the source edge $\mathsf{origin}(\zeta) = (1, \mathrm{German}, 0)$ to the view graph.

In the following, although the trace t can be used for both nodes and edges, we only focus on edges instead of nodes, and edge renamings as the update operations. However, the node trace can be used to support insertion operation by highlighting the node in the source on which a graph corresponding to the graph to be inserted in the view will be attached. For the deletion operation, we can relatively easily extend the data structure of the trace defined in the previous section to be tree-structured, being able to trace multiple sources, and extend rule (Rec) to associate each edge created by the body expression with the trace of the edge bound to the label variable.

For editability analysis, the following notion of equivalence is used. Given the partial function $\mathsf{origin}_{\mathrm{E}} : \mathit{Edge} \to \mathit{Edge}$ defined by $\mathsf{origin}_{\mathrm{E}}\ \zeta = \mathsf{origin}(\zeta)$ if $\mathsf{origin}(\zeta) \in \mathit{Edge}$, or undefined otherwise. Then, view edges ζ_1 and ζ_2 are equivalent, denoted by $\zeta_1 \sim \zeta_2$, if and only if $\mathsf{origin}_{\mathrm{E}}\, \zeta_1 = \mathsf{origin}_{\mathrm{E}}\, \zeta_2$. All edges for which $\mathsf{origin}_{\mathrm{E}}$ is undefined are

considered equivalent. The equivalence class for edge ζ is denoted by $[\zeta]_\sim$. We define our updates as $\mathsf{upd} : \mathit{Upd}$ where $\mathit{Upd} = \mathit{Edge} \to \mathit{Label}$, an expression and its environment at position p as $\mathsf{expr} : \mathit{Pos} \to \mathit{Expr} \times \mathit{Env}$, and update propagation along trace t as $\mathsf{prop} : \mathit{Env} \to \mathit{Upd} \to \mathit{Env}$. Then, our editability analysis is defined as follows.

Definition 1 (Editability checking)**.** *Given $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ and update upd on g, editability checking succeeds if all the following three conditions are satisfied.*

1. *For all updated edges ζ, other view edges in $[\zeta]_\sim$ are unchanged or consistently updated, i.e., $\forall\zeta \in \mathrm{dom}(\mathsf{upd}).\forall\zeta' \in [\zeta]_\sim.\zeta' \notin \mathrm{dom}(\mathsf{upd}) \vee \mathsf{upd}(\zeta) = \mathsf{upd}(\zeta')$.*
2. *For every $\mathbf{if}\ e_\mathrm{B} \ldots$ expression in the backward evaluation path, applying the edit to the binding does not change the condition, i.e., $\forall p \in \{\cup[p \mid p \in t(\zeta), p \in \mathit{Pos}] \mid \zeta \in \mathrm{dom}(\mathsf{upd})\}, \mathsf{expr}(p) = (\mathbf{if}\ e_\mathrm{B} \ldots, \rho), \mathcal{F}[\![e_\mathrm{B}]\!]_\rho = \mathcal{F}[\![e_\mathrm{B}]\!]_{\mathsf{prop}(\rho,\mathsf{upd})}$. For the case of label variables, this interference can be checked by $\$l \in \mathcal{FV}(e_\mathrm{B}), \rho'(\$l) = (\_,\tau), \mathsf{last}(\tau) = \zeta'$ for $\zeta' = \mathsf{origin}_\mathrm{E}\,\zeta$. Weakened version for graph variables is: $\forall\$g \in \mathcal{FV}(e_\mathrm{B}).(g',t') = \rho'(\$g), \forall\zeta'' \in g'.\mathrm{E}.\,\mathsf{last}(t'((\zeta'')) \neq \mathsf{origin}_\mathrm{E}(\zeta)$ for all $\zeta \in \mathrm{dom}(\mathsf{upd})$.*
3. *No edited edge trace to code position, i.e., $\forall\zeta \in \mathrm{dom}(\mathsf{upd}).\,\mathsf{origin}(\zeta) \notin \mathit{Pos}$.*

Further, we recap all the three run-time errors to reject updates [13] as (1) failure at environment merging $\uplus$ ("inconsistent change detected"), (2) failure at $\mathcal{B}[\![\mathbf{if}\ldots]\!]$ ("branch behavior changed") and (3) failure at $\mathcal{B}_\mathrm{L}[\![\mathtt{a}]\!]$("no modifications allowed for label literals."). Then the following lemmas hold.

Lemma 1. *Condition 1 in Def. 1 is false iff error (1) occurs.*

Lemma 2. *Error (2) implies condition 2 in Def. 1 is false.*

Lemma 3. *Condition 3 in Def. 1 is false iff error (3) occurs.*

Based on these lemmas, we have the following theorem.

Theorem 1 (Soundness of editability analysis)**.** *Given forward transformation $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ and update upd on g to g', success of editability checking in Def. 1 implies $\mathcal{B}[\![e]\!](\rho_0, g')$ defined and results in $\mathsf{prop}(\rho_0, \mathsf{upd})$.*

Thus edit on the view edge ζ with $\zeta' = \mathsf{origin}_\mathrm{E}\,\zeta$ defined is propagable to ζ' in the source graph by $\mathcal{B}[\![]\!]$, when the checking in Def. 1 succeeds. An edit on the view edge ζ with $\mathsf{origin}(\zeta) = p \in \mathit{Pos}$ is not propagable to the source by Lemma 3. Editing label constant at p in the transformation would achieve the edits, with possible side effects through other copies of the label constant.

Consider the example in Listing 1 with the source graph of Fig. 1 and view graph of Fig. 2. We get four equivalence classes, one each for the source edges $(1, \mathtt{German}, 0)$, $(3, \mathtt{Europe}, 2)$, $(5, \mathtt{Austrian}, 4)$ and $(11, \mathtt{German}, 10)$, as well as the class that violate condition 3. For view edge $\zeta = (3, \mathtt{German}, 1)$, we have $(4, \mathtt{German}, 2) \in [\zeta]_\sim$ via $\mathsf{origin}_\mathrm{E}\,\zeta = (1, \mathtt{German}, 0)$, so these equivalent edges can be selected simultaneously, inconsistent edits on which can be prevented (Lem. 1). Direct edits of the view edge $\zeta = (0, \mathtt{result}, 14)$ are suppressed since $\mathsf{origin}\,\zeta \in \mathit{Pos}$ (Lem. 3). On condition 2, when the view edge $\zeta = (7, \mathtt{Europe}, 5)$ is given, all the code positions in t ζ'' for $\zeta'' \in {\mathsf{origin}_\mathrm{E}}^{-1}(\mathsf{origin}_\mathrm{E}\,\zeta)$ are checked if the positions represent conditionals that refer variables, change of those binding would change the conditions and would be rejected by $\mathcal{B}$. We obtain the position for variable reference $\$l$ in the condition $(\$l = \mathtt{Europe})$ for warning.

Soundness Proof of the Editability Analysis

Cases in which completeness is lost: Note that system may choose the weakened version of condition 2, and just issue warning (i.e., whenever the edge being updated has common source edge with any variable reference occurring free in any of the conditional expressions along the execution path), because when condition includes not only the simple label comparison expressions but also graph emptiness checking by $\mathsf{isEmpty}$, the cost of fully checking the change of the condition may amount to re-executing the most part of the forward transformation. If this warning-only strategy is chosen, backward transformation may succeed despite this warning, because the warning does not necessarily mean condition value changes. Therefore, the checking is not complete (may warn some safe updates). We argue that this "false alarm" is not so problematic in practice, with the following reasons. The argument of $\mathsf{isEmpty}$ usually includes **select** to test an existence of certain graph pattern. So we further analyze conditions in **select** down to simple label equivalence check. If the argument is a bare graph variable reference, then we analyze its binder. If the graph variable is $\$db$ (the entire input graph), then any changes

cause false alarms. This predicate may be used only for debugging purpose to exclude empty source graphs, but otherwise we don't think we frequently encounter this situation.

Now we provide a proof sketch. First, we prove Lemma 3. We do this by parallel case analysis on augmented forward semantics and backward semantics [13].

Base case No edit is possible for $\{\}, ()$ and $\&y$ that produce no edge. For a transformation $\{\mathtt{a} : e'\}^p$ and the view graph as $v \xrightarrow{\mathtt{a}} G$, since $\mathtt{a}$ is a constant, augmented forward semantics generates code position p as trace, which violates condition 3 when $\mathtt{a}$ is being updated. Backward semantics rejects with error (3).

Inductive case

Transformation $\{\$l : e'\}^p$ and the view graph as $v \xrightarrow{l} G$: Update on l to l' will be handled by the backward semantics of $\$l$, which updates its reference. Suppose it is introduced by an expression $\mathbf{llet}\ \$l = \mathtt{a}\ \mathbf{in}\ \{\$l : e'\} \ldots$. Then binding $\$l \mapsto l'$ is generated and it is passed to the backward evaluation of label expression $\mathtt{a}$, which is constant. So no value other than $\mathtt{a}$ is allowed. So backward transformation fails with error (3). The trace generates code position of label constant $\mathtt{a}$, which also signals violation of condition 3. If $\$l$ is bound by $\mathbf{rec}(\lambda(\$l, \_).\{\$l : e_1\})(e_2)$, then the update is also translated to the argument expression e_2, so if corresponding part in e_2 originates from a label constant, then the condition is violated and backward transformation fails similarly.

$e_1 \cup e_2, e_1 \oplus e_2, e_1 \mathbin{@} e_2$: We reduce the backward transformation to that of subexpressions, assuming decomposition operation for each operator. So assuming Lemma 3 for these subexpressions, Lemma 3 holds for entire expression. Similar argument applies for $\mathbf{cycle}(e)$, $\&\_ := e$ and $\mathbf{if}\ \_\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$.

$\mathbf{rec}(e_1)(e_2)$: What is produced as a trace depends on e_1 and e_2. Assuming the decomposition of target graph, the backward evaluation is reduced to those of e_1 and e_2. The only interesting case is a graph variable reference as (sub) expression. The update on the target graph that was produced by the variable reference is reduced to updated graph variable bindings, and the bound graph is aggregated for each edge in the input graph that was produced by e_2, and passed to the backward evaluation of e_2. The trace, on the other hand, is produced by e_2 and combined with the trace by e_1. If e_1 is a graph variable reference, then the (sub) trace induced from the trace from e_2 is used, so the edge that trace back to constant is inherited in the trace of the result of **rec**, so attempt to update an edge that traces back to the constant produced by e_2 and carried by the graph variable reference is signaled with condition 3. So, assuming Lemma 3 on e_2 by induction hypothesis, Lemma 3 holds for the entire expression.

Thus conclude the proof for Lemma 3.

Next we prove Lemma 1. When violation of 1 is signaled, multiple edge labels with the same equivalence class is updated to different values. The proof can be conducted similarly to the case for Lemma 3, except that we focus on multiple edge labels generated by different subexpressions of the transformation.

$\{\$l : \$g\}$: Suppose bindings $\$l \mapsto \mathtt{a}$ and $\$g \mapsto \{\mathtt{b} : G\}$ are generated by updates on the target graph. Further, suppose these bindings are generated by $\mathbf{rec}(\lambda(\$l', \_).\, \mathbf{rec}(\lambda(\$l, \$g).\{\$l : \$g\})(\{\$l' : \{\$l' : \{\}\}\}))(\$db)$ as the inner body expression of **rec** and $\$db$ is bound to graph $\{\mathtt{x} : \{\}\}$. Then, backward evaluation of the argument expression in the inner **rec**, i.e., $\{\$l' : \{\$l' : \{\}\}\}$ will produce the bindings $\$l' \mapsto \mathtt{a}$ and $\$l' \mapsto \mathtt{b}$ for the first label expression $\$l'$ and the label construction expression $\{\$l' : \{\}\}$, respectively, and merge these bindings by $\uplus$ operator relative to original binding $\$l' \mapsto \mathtt{x}$. Then backward transformation fails because of the conflicting bindings. In this case, the augmented forward semantics also allocate the top and the following edge the same origin edge, so the update signals violation of condition 1. So violation of condition 1 coincides failed backward transformation. Similar situation can be observed for graph constructors $\cup, \oplus$ and $@$ that unifies the graphs, when conflicting updates are attempted to edges originated from identical edges. The merge phase for the subexpressions in the backward transformation causes the failure. Note that the duplicate is propagated through variable bindings, so conflicting updates can be detected within the target graph created by a single graph variable reference, like $\mathbf{let}\ \$g' = \mathbf{rec}(\lambda(\$l', \_).\, \mathbf{rec}(\lambda(\$l, \$g).\{\$l : \$g\})(\{\$l' : \{\$l' : \{\}\}\}))(\$db)\ \mathbf{in}\ \$g'$. The failure takes place in the **let** expression in this case. We omit the detailed case analysis here.

For the Lemma 2, its proof is straightforward by the warning algorithm in the long version. Note that if we exclude the update on target that causes violation of condition 2, we have soundness and completeness. In general, we only have soundness. This concludes the proof sketch. □

6 Related Work

Our novelty is to analyze editability using traces in compositional bidirectional graph transformations.

Tracing mechanism. Traceability is studied enthusiastically in model-driven engineering (MDE) [9, 25]. Van

Amstel et al. proposed a visualization framework TraceVis for chains of ATL model transformations [30]. Systematic augmentation of the trace-generating capability with model transformations [22] is achieved by higher-order model transformations [29]. Although they also trace between source, transformation and target, we also use the trace for editability analysis. We can help improve this kind of unidirectional tool. Grain size of transformation trace in TraceVis is at ATL rule level, while our trace is more fine grained. If TraceVis refines the transformation trace to support inspection of guards (conditions), they can also support control flow change analysis. To do so, ATL engine may be extended to maintain the variable binding environment similar to ours at run-time, and use it to trace bindings within the guards.

Our own previous work [13] introduced the trace generation mechanism, but the main objective was the bidirectionalization itself. The notion of traces has been extensively studied in a more general context of computations, like provenance traces [4] for the nested relational calculus.

Triple Graph Grammars (TGG) [26] and frameworks based on them are studied extensively and are applied to MDE [1] including traceability [11]. They are based on graph rewriting rules consisting of triples of source and target graph pattern, and the correspondence graph in-between which explicitly contains the trace information. Grammar rules are used to create new elements in source, correspondence and view graphs consistently. By iterating over items in the correspondence graph, incremental TGG approaches can also work with updates and deletions of elements [10]. The transformation language UnQL is compositional in that the target of a (sub)transformation can be the source of another (sub)transformation, while TGG is not. Our tracing over compositions are achieved by keeping track of variable bindings.

Xiong et al. [32]'s traces are editing operations (including deletion, insertion and replacement) embedded in model elements. They are translated along with transformation (in ATL bytecode). They check control flow changes in these embedded operations *after* finishing backward transformations and reject updates when these changes are detected. ATL level trace could have been computed using our approach for correspondence analysis.

Target Element Classification and Editability Analysis. Model element classification has been studied in the context of Precedence Triple Graph Grammars [23]. Though the motivation is not directly related to ours, their classification is indeed similar in the sense that both group elements in graphs. The difference is that the equivalence class in [23] is defined such that all the elements in the class are affected in a similar manner (e.g., created simultaneously) while in our equivalence class, members are defined only on the target side, and they are originated from the same source edge, so edits on any member result in edits on the source edge. The equivalence class in [23] may correspond to the subgraph created by the same "bulk" by the body expression of **rec** with respect to common edge produced by the argument expression, so that when the common edge is deleted, then all the edges in the bulk are deleted simultaneously.

Another well-studied bidirectional transformation framework called semantic bidirectionalization [31] generates a table of correspondence between elements in the source and those in the target to guide the reflection of updates over *polymorphic* transformations, without inspecting the forward transformation code (thus called semantic). The entries in the target side of the table can be considered as equivalence classes to detect inconsistent updates on multiple target elements corresponding identical source element. Although UnCAL transformations are not polymorphic in general because of the label comparison in the **if** conditionals with constant labels, prohibiting the semantic bidirectionalization approach. Matsuda and Wang [24] relaxed this limitation by run-time recording and change checking of the branching behaviors to reject updates causing such change. They also cope with data constructed during transformation (corresponding to constant edges in our transformation). So our framework is close to theirs, though we utilize the syntax of transformation. We can trace nodes (though we focused on edge tracing in this paper) while theirs cannot.

Hu et al. [21] treat duplications explicitly by a primitive Dup. They do not highlight duplicates in the view, rather they rely on another forward transformation to complete the propagation among duplicates.

7 Conclusion

In this paper, we proposed, within a compositional bidirectional graph transformation framework based on structural recursion, a technique for analyzing the correspondence between source, transformation and target as well as to classifying edges according to their editability. We achieve this by augmenting the forward semantics with explicit correspondence traces. The incorporation of this technique into the GUI enables the user to clearly visualize the correspondence. Moreover, prohibited edits such as changing a constant edge and updating a group of edges inconsistently are disabled. This allows the user to predict violated updates and thus do not attempt them at the first place. In this paper we only focused on edge-reaming as edit operation but the proposed

framework can also support edge-deletion by slight extension, while insertion handling can be supported by the present node tracing.

As a future work, we would like to utilize the proposed framework to optimize the performance of edge deletion and subgraph insertion using the backward transformation semantics for in-place update semantics, because we currently handle these update operations by different algorithms. In particular, insertions use a general inversion strategy which is costly. We already have limited support in this direction, however we are still to establish formal bidirectional properties for complex expressions. Leveraging the traces in the forward semantics indicating which edge is involved in the branching behavior, we could safely determine the part that accepts the insertion or deletion of that part reusing the in-place update semantics, thus achieving "cheap backward transformation". The proposed mechanism can be used for any other trace-based approaches, unidirectional or bidirectional, as we discussed in Section 6, and we would like to pursue this direction as well.

Acknowledgments The authors would like to thank Zhenjiang Hu, Hiroyuki Kato and other IPL members, and AtlanMod team for their valuable comments. We also thank the reviewers for their constructive feedbacks. The project was supported by the International Internship Program of the National Institute of Informatics.

References

[1] Carsten Amelunxen, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08*, pages 807–810. ACM, 2008.

[2] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[4] James Cheney, Umut A. Acar, and Amal Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT'09*, pages 260–283, 2009.

[6] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *MODELS'11*, volume 6981 of *LNCS*, pages 304–318. 2011.

[7] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Uncertainty in bidirectional transformations. In *Proc. 6th International Workshop on Modeling in Software Engineering (MiSE)*, pages 37–42. ACM, 2014.

[8] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[9] Ismenia Galvao and Arda Goknil. Survey of traceability approaches in model-driven engineering. In *EDOC '07*, pages 313–324. IEEE Computer Society, 2007.

[10] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, 2008.

[11] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Inter-modelling: From theory to practice. In *MODELS'10*, pages 376–391. Springer-Verlag, 2010.

[12] Soichiro Hidaka, Martin Billes, Quang Minh Tran, and Kazutaka Matsuda. Trace-based approach to editability and correspondence analysis for bidirectional graph transformations. Technical report, May 2015. `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/tesem.pdf`.

[13] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP'10*, pages 205–216. ACM, 2010.

[14] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, Keisuke Nakano, and Isao Sasano. Marker-directed optimization of UnCAL graph transformations. In *LOPSTR'11, Revised Selected Papers*, volume 7225 of *LNCS*, pages 123–138, July 2012.

[15] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations (short paper). In *ASE'11*, pages 480–483. IEEE, 2011.

[16] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. *Progress in Informatics*, (10):131–148, March 2013. Journal version of [15].

[17] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards compostional approach to model transformations for software development. Technical Report GRACE-TR08-01, GRACE Center, National Institute of Informatics, August 2008.

[18] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. A compositional approach to bidirectional model transformation. In *ICSE New Ideas and Emerging Results track, ICSE Companion*, pages 235–238. IEEE, 2009.

[19] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards a compositional approach to model transformation for software development. In *Proc. of the 2009 ACM symposium on Applied Computing (SAC)*, pages 468–475. ACM, 2009.

[20] Soichiro Hidaka and James F. Terwilliger. Preface to the third international workshop on bidirectional transformations. In *Workshops of the EDBT/ICDT 2014*, pages 61–62, 2014.

[21] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04*, pages 178–189, 2004.

[22] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 29–37, 2005.

[23] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient model synchronization with precedence triple graph grammars. In *ICGT'12*, pages 401–415. Springer-Verlag, 2012.

[24] Kazutaka Matsuda and Meng Wang. "bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 2014. DOI:10.1016/j.scico.2014.07.008.

[25] Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Softw. Syst. Model.*, 10(4):469–487, October 2011.

[26] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94*, volume 903 of *LNCS*, pages 151–163, June 1995.

[27] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

[28] Perdita Stevens. Bidirectionally tolerating inconsistency: Partial transformations. In *FASE'14*, volume 8411 of *LNCS*, pages 32–46, 2014.

[29] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 18–33, 2009.

[30] Marcel F. van Amstel, Mark G. J. van den Brand, and Alexander Serebrenik. Traceability visualization in model transformations with TraceVis. In *ICMT'12*, pages 152–159, 2012.

[31] Janis Voigtländer. Bidirectionalization for free! (pearl). In *POPL '09*, pages 165–176. ACM, 2009.

[32] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173, November 2007.

Towards a Principle of Least Surprise for bidirectional transformations

James Cheney[1], Jeremy Gibbons[2], James McKinna[1], and Perdita Stevens[1]

1. School of Informatics
University of Edinburgh, UK
Firstname.Lastname@ed.ac.uk

2. Department of Computer Science
University of Oxford, UK
Firstname.Lastname@cs.ox.ac.uk

Abstract

In software engineering and elsewhere, it is common for different people to work intensively with different, but related, artefacts, e.g. models, documents, or code. They may use bidirectional transformations (bx) to maintain consistency between them. Naturally, they do not want their deliberate decisions disrupted, or their comprehension of their artefact interfered with, by a bx that makes changes to their artefact beyond the strictly necessary. This gives rise to a desire for a principle of Least Change, which has been often alluded to in the field, but seldom addressed head on. In this paper we present examples, briefly survey what has been said about least change in the context of bx, and identify relevant notions from elsewhere that may be applicable. We identify that what is actually needed is a Principle of Least Surprise, to limit a bx to reasonable behaviour. We present candidate formalisations of this, but none is obviously right for all circumstances. We point out areas where further work might be fruitful, and invite discussion.

1 Introduction

Bx restore consistency between artefacts; this is the primary definition of what it is to be a bx. In broad terms, defining a bx includes specifying what consistency should mean in a particular case, though this may have a fixed, understood definition in the domain; it also involves specifying how consistency should be restored, where there is a choice. In interesting cases, the bx does not just regenerate one artefact from another. This is essential in situations where different people or teams are working on both artefacts, otherwise their work would be thrown away every time consistency was restored. Formalisms and tools differ on what information is taken into account: is it just the current states of the two models (state-based or relational approach: we will use the latter term in this paper) or does it also include intensional information about how they have recently changed, about how parts of them are related, etc.? A motivation for including such extra information, even at extra cost, is often that it enables bx to be written that have more intuitively reasonable behaviour. We still lack, however, any bx language in mainstream industrial use. The reasons for this are complex, but we believe a factor is that it is not yet known how best to provide reasonable behaviour at reasonable cost, or even what "reasonable" should mean.

Even in the relational setting, the best definition of "reasonable" is not obvious. Meertens' seminal but unpublished paper [14] discussed the need for a Principle of Least Change for constraint maintainers, essentially what we now call relational bx. His formulation is:

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

> The action taken by the maintainer of a constraint after a violation should change no more than is needed to restore the constraint.

It turns out, however, that there are devils in the details.

We think that making explicit the least change properties that one might hope to achieve in a bx is a step towards guiding the design and use of future languages. In the long term, this should support bx being incorporated into software development in such a way that they do not violate the "principle of least surprise" familiar from HCI: their developers and their users should be able to rely on bx behaving "reasonably". If a community could agree on a precise version of a least change principle that should always be obeyed, we might hope to develop a bx language in which *any* legal bx would satisfy that principle (analogous to "well typed programs don't go wrong"). If that proves too ambitious – perhaps there is a precise version of the principle that is agreed to be desirable, but which must on occasion be disobeyed by a reasonable bx – then a fall-back position would be to develop analysis tools that would be capable of flagging when a particular bx was at risk of disobeying the principle. It could then be scrutinised particularly closely e.g. in testing; or perhaps a bx engine would provide special run-time behaviour in situations where it might otherwise surprise its users, e.g. asking the user to make or confirm the selection of changes, while a "safer" change propagation could be made without interaction.

We find it helpful to bear in mind the domain of model-driven development (MDD) and how bx fit into it. Hence in this paper we will refer to our artefacts as "models", using the term inclusively, and we will use the term "model space" for the collection of possible models, with any structure (e.g. metric) it may have. The fundamental idea of MDD is that development can be made more efficient by allowing people to work with models that express just what they need to know to fulfill their specific role. The reason this is helpful is precisely that it avoids distracting – surprising – people by forcing them to be aware of information and *changes* that do not affect them. The need for bx arises because the separation between relevant information (should be in the model) and irrelevant information (should not be in the model) often cannot be made perfectly: changes to the model are necessitated by changes outside it. This is inherently dangerous, because the changes are, to a degree, inherently surprising: they are caused by changes to information the user of the model does not know about.

Without automated bx, these changes have to be decided on by humans. Imagine I am at a developers' meeting focused (solely) on deciding how to restore consistency by changing my model. There might well be disagreement about the optimal way to achieve this end. Some courses of action, however, would be considered unreasonable. For example, if I have added a section to my model which is (agreed to be) irrelevant to the question of whether the models are consistent, then to delete my new section as part of the consistency restoration process would plainly be not only suboptimal, but even unreasonable. As well as disagreement about what course of action was *optimal*, we might also expect disagreement also about what courses were *(un)reasonable*; there might be discussion, for example, about trade-offs between keeping the changes small, and keeping them easy to describe, or natural in some sense, or maintaining good properties of the models.

Humans agree what is reasonable by a social process; to ensure that a bx's behaviour will be seen by humans as reasonable, we must formalise the desired properties. We will bear in mind the meeting of reasonable developers as a guide: in particular, it suggests that we should not expect to find one right answer.

One property is (usually, but not always!) easily agreed: if the models are already consistent, the meeting, or bx, has no job to do. It will make no change. This is hippocraticness.

Going beyond this, the meeting attempts to ensure that changes made in order to restore consistency are not more disruptive to the development process than they need to be. For a given informal idea of the "size" of a change to a model, it does not necessarily follow that two changes of that size will be equally disruptive. For example, a change imposed on a part of a model that its developers had thought they understood well is likely to be more disruptive than a change imposed on a part that they did not know much about, e.g. a placeholder; and a change to something the developers felt they owned exclusively is likely to be felt as more disruptive than a change to something that they felt they shared. We see the need for something more like "least disruption" or "least surprise" than "least change".

It would be easy to throw up one's hands at the complexity of this task. Fortunately, the study of bx is far from the only situation in which such considerations arise, and we are able to draw on a rich literature, mathematical and otherwise.

1.1 Notation and scope

When talking about state-based relational bx we use the now-standard (see e.g. [15]) notation $R : M \leftrightarrow N$ for a bx comprising consistency relation R and restorers $\overrightarrow{R} : M \times N \to N$, $\overleftarrow{R} : M \times N \to M$. Such bx will be assumed to be correct (restorers do restore consistency) and hippocratic (given consistent arguments, they make no change). Where possible we will broaden the discussion to permit, for example, intensional information about changes to models, and auxiliary structures besides the models themselves. However, even then, we will assume that consistency restoration involves taking one model as authoritative, and changing the other model (and, perhaps, some auxiliary structures). More complex scenarios are postponed to future work.

Since consistency is primary, we will assume that the bx developer defines what it is for models to be consistent. It may be simply a relation over a pair of sets of models, or it may be something more complex involving witnessing structures such as sets of trace links. Regardless, we assume that we do not have the right to change the developers' notion of what is a good, consistent state of the world. Our task is to constrain how a bx restores the world to such a state. Of course, in general there are many ways to do so: this is the essence of the problem. In a setting where, given one model, there is a unique choice of consistent other model, there is no role for a Least Change Principle, because there is no choice between consistent resolutions to be made. Even here, though, we might wish to know whether a specific principle was obeyed, in order to have an understanding of whether developers using it were likely to get nasty surprises. This begins the motivation for replacing a search for a Least *Change* Principle with a broader consideration of possible Least *Surprise* Principles.

Let us go on to make explicit a key non-ambition of this work: we do not seek to identify a principle that is so strong that, given the consistency relation, there is a unique way to restore consistency that satisfies it. We envisage that the bx programmer has to specify consistency and may also have to provide information about how consistency is restored where there is a choice; we also wish to leave open the door to applying the chosen Principle to effectful bx [1], e.g. bx that involve user interaction, where the user might also be involved in this choice (in a more sophisticated way than having to choose between all consistent models). By contrast, some of the work we shall draw on ([8] for example) does regard it as a deficiency in the principle if there might be more than one way to satisfy it.

We will use a few basic notions of mathematical analysis (see e.g. [17]), such as

Definition A *metric* on a set X is a function $d : X \times X \to \mathbb{R}$ such that for all $x, y, z \in X$:

1. $d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, y) + d(y, z) \geq d(x, z)$

1.2 Structure of the paper

In Sections 2 and 3 we address two major dimensions on which possible principles may vary. The first dimension concerns what structure is relevant to the relative cost we consider a change to have. This dimension has had considerable attention, but we think it is worth taking a unified look. The second, concerning whether we offer guarantees when concurrent editing has taken place, seems to be new, but based on our investigations we think it may be important. In Section 4 we collect some examples that we shall use in the rest of the paper; the reader may not wish to read them all in detail until they are referred to, but, as many are referred to from different places, we have preferred to place them together. In Section 5 we consider approaches based on ordering changes themselves; we go on in Section 6 to consider perhaps the purest form of "least change", based on metrics on the model spaces. Because of the disadvantages inherent in that approach, we decide to go on, in Section 7, to consider in some detail an approach based on guaranteeing "reasonably small" changes instead. Section 8 considers categorical approaches, and Section 9 concludes and discusses some future work that has not otherwise been mentioned. A major goal of the paper is to inspire future work, though, and we point out areas that might repay it throughout. We do not have a Related Work section, because the entire paper is about related work. Our own modest technical contribution is in Section 7.

2 Beyond changes to models: structure to which disruption might matter

How should one change to a model be judged larger than another? Typically it is not hard to come up with a naive notion based on the presentation of the model, but this can mislead. We can sometimes explain the phenomenon that "similarly sized" changes to a model are not equally disruptive, by identifying (making explicit

in our formalism) auxiliary structure which changes more in one case than the other: the reason one change feels more disruptive than another is because it disrupts the auxiliary structure more, even if the disruption to the actual model is no larger. In the database setting, Hegner's information ordering [8] makes explicit *what is true about* a value. A change to a model which changes the truth value of more propositions is considered larger. Hegner's auxiliary structure does not actually add information: it can be calculated from the model. However, his setting had an established notion of change to a model (sets of tuples added and dropped, with supersets being larger changes). Thus, although we might wonder about redefining the "size" of changes so that changes that changed more truth values would be considered larger, that would not have been an attractive option in his setting. In MDD the idea of preferring changes that make less difference to what the developer thinks they know is attractive, but things are (as always!) more blurred: what is known, let alone knowable, about part of a model may not be deducible from the model, and there is no commonly agreed notion of change. Perhaps future bx language developers should explore, for example, weighting classes by how often their names appear in documentation, and/or allowing the bx user to assign weights, and using the weights in deciding on changes.

Another major class of auxiliary structure to which disruption may matter, which definitely does involve information outside the model itself, is witness structures: structures whose intention is to capture something about the relationship between the models being kept consistent. A witness structure witnessing the consistency of two models m and n is an auxiliary structure that may help to demonstrate that consistency. In the simplest setting of relational bx, the unique witness structure relating m and n is just a point, and the witness structure in fact carries no extra information beyond the consistency relation itself ("they're consistent because I say so"). In TGGs, the derivation tree, or the correspondence graph, may be seen as witness structures ("they're consistent because this is how they are built up together using the TGG rules"). In an MDD setting, a witness structure may be a set of traceability links that helps demonstrate the consistency by identifying parts of one model that go with parts of another ("they're consistent because this part of this links to that part of that"). In [15] some subtle issues were discussed concerning whether the links that demonstrate consistency embodied in QVT transformations should be followable in only one direction or both, and the paper also presented a game whose winning strategies can be seen as richer witness structures ("they're consistent because here's how Verifier wins a consistency game on them"). A witness structure could even be a proof. ("They're consistent because Coq gave me this proof that they are.")

Thus many types of witness structures can exist, and even within a type, different witness structures might witness the consistency of the same pair of models. (A dependently typed treatment is outside the scope of this paper, but is work in progress.) The extent to which the witness structure is explicit in the minds of the developers who work with the models probably varies greatly – indeed, future bx language designers should consider the implications of this (cf considering the user's mental model, in UI design). Care will be needed to ensure practical usability of change size comparisons based on witness structures, but at least there is some hope that we can do better with than without them.

Fortunately, for our purposes, it suffices to observe that much of what we say in a relational setting can be "lifted" to a setting with auxiliary structures; the developer's modification is still to a model, but the bx may in response modify not just the other model but also the auxiliary structure, and when we compare or measure changes we may be using the larger structure. We leave the rest of this fascinating topic for future work.

3 Beyond least change: weak and strong least surprise

Strictly following a formalisation of a "least change" definition, such as Meertens suggested, allows the bx writer very limited freedom to choose how consistency restoration is done: the definition prescribes *optimal* behaviour by a rigid definition of optimal, and the only real flexibility the user has concerns the way the size of a change is measured. In the spirit of the "least surprise" principle from UI design, it may be more fruitful to formalise what behaviour is *reasonable*; this gives more flexibility to the bx writer, while still promising absence of surprise. Here, reasonableness will be about the sizes of changes: we will set the scene for continuity-based ideas, guaranteeing that the size of the change to a model caused by consistency restoration can be limited, in terms of the size of the change being propagated.

When should such guarantees apply? Does the bx guarantee to behave reasonably even when both models have been modified concurrently? We illustrate this with informal scenarios in the relational setting.

Suppose there are two users, Anne working with models from set A and Bob working with model set B. Suppose Anne has made a change she regards as small. We would like to be able to guarantee, by restricting to bx with a certain good property, that the *difference* this change makes to Bob is small. The intuition is

that Anne is likely to consider a large change much more carefully than a small change. We do not wish a not-very-considered choice by Anne (e.g. adding a comment) to have a large unintended consequence for Bob. (If Bob changes his model and Anne's must be modified to restore consistency, everything is dual.)

Weak least surprise guarantees that the change from b to $b' = \overrightarrow{R}(a', b)$ is small, *provided that a and b are consistent* and that the change from a to a' is small. That is, it supports the following scenario:

- models a and b are currently consistent;
- now Anne modifies her model, a, by a small change, giving a'
- then we restore consistency, taking Anne's model as authoritative; that is, we calculate $b' = \overrightarrow{R}(a', b)$ and hand it to Bob to replace his current model, b
- Bob's model has suffered a small change, from b to b'.

That is, the guarantee we offer operates under the assumption that Bob's model hasn't changed, since it was last consistent with Anne's, until we change it. If Bob continues to work on it in the meantime, all bets are off. We guarantee nothing if the two models have been edited concurrently.

Strong least surprise imposes a more stringent condition on the bx and offers a correspondingly stronger guarantee to the bx users. It allows that Bob may be working concurrently with Anne; his current model, b, might, therefore, not be consistent with Anne's current model a. Here the guarantee we want is that, provided the change Anne makes from a to a' is small, the choice of whether to restore consistency before or after making this small change will make only a small difference to Bob, regardless of what state his model is in at the time. It supports the scenario:

- we don't know what state Bob's model is in with respect to Anne's – both may be changing simultaneously
- Anne thinks about changing hers a little, and isn't sure whether or not to do so (e.g., she dithers between a and a' which are separated by a small change);
- the difference between
  - the effect on Bob's model if Anne does decide to make her small change and
  - the effect on Bob's model if she doesn't,

  is small, regardless of what state Bob's model is in when we do restore consistency. In the state-based relational setting, this amounts to saying that *for any* b the change from $\overrightarrow{R}(a, b)$ to $\overrightarrow{R}(a', b)$ can be guaranteed to be small, provided the change from a to a' is small. Note that both $\overrightarrow{R}(a, b)$ and $\overrightarrow{R}(a', b)$ might be a "large change" from b; to demand otherwise would be unreasonable, as a and b, being arbitrary, might be very seriously inconsistent. We can ask only that these results are a small change from one another.

Weak least surprise is a weakening of strong least surprise because by hippocraticness, if $R(a, b)$ then $\overrightarrow{R}(a, b) = b$ so we get the weak definition by instantiating the strong definition at models b that are consistent with a, only.

Strong and weak least change vary in the condition that is placed on the initial state of the world, before we will know that a small change on one side will cause only a small change on the other: does the condition hold for all $(a, b) \in A \times B$, or only for $(a, b) \in C \subseteq A \times B$? In the weak variant C is the consistent pairs.[1] Alternatively, we could take $C = \{(a, b) : b \in N(a)\}$ where for any a, $N(a)$ is the models that are, maybe not consistent with, but at least reasonably sane with respect to, a. Another promising option is $C = M \times N$ for a subspace pair [16] (M, N) in (A, B). A subspace pair is a place where the developers working with both models can, jointly, agree to stay, in the sense that if they do not move their model outside the subspace pair, the bx will not do so when it restores consistency. Imposing least surprise would guarantee, additionally, that small changes on one side lead to small changes on the other. These variants might repay further study, particularly in that identifying "good" parts of the pair of model spaces might be possible even if inevitably the bx must have bad behaviour elsewhere. Example 4.7 illustrates.

4 Examples

In this section we present a number of examples intended to be thought-provoking, in that they demonstrate various ways in which it can fail to be obvious what consistency restoration best obeys a Least Surprise Principle. Some are drawn from the literature. We have collected them in one section for ease of reference; the reader should feel free to skim and return. Names are those used in the Bx Examples Repository[2] [4].

[1] Cf the distinction between undoability and history ignorance.

[2] http://bx-community.wikidot.com/examples:home

Let us begin with an MDD-inspired example, which illustrates that we may not always want the consistent model which is intuitively closest.

Example 4.1 (ModelTests) *Suppose bx R relates UML model m with test suite n, saying that they are consistent provided that every class in m stereotyped $\langle\langle$persistent$\rangle\rangle$ has a test class of the same name in n, containing an appropriate (in some specified sense) set of tests for each public operation, but n may also contain other tests. You modify the test class for a $\langle\langle$persistent$\rangle\rangle$ class C, to reflect changes made in the code to the signatures of C's methods, e.g., say* `int` *has changed to* `long` *throughout. R now propagates necessary changes to the model m. You probably expect R to perform appropriate changes to the detail of persistent class C in the model, changing* `int` *to* `long` *in the signatures of its operations. However, a different way to restore consistency would be to remove the stereotype from C, so that there would no longer be any consistency requirements relating to C; this probably involves a shorter edit distance, but not what is wanted.*

There are two possible reactions to this. One is to argue that the desired change to the detail of C should satisfy a good Principle of Least Change, and that if it does not, we have the wrong Principle. We may take this as a justification for considering the size of changes made to auxiliary structures as well as the size of changes made to the model itself. We might imagine a witness structure comprising trace links between public operations of $\langle\langle$persistent$\rangle\rangle$ classes and the corresponding tests. In this case, we might argue that although removing the stereotype from C is a small change to the UML model, it involves removing a trace link for every public operation of C from the witness structure; this might be formalised to justify an intuitive feeling that actually removing the stereotype is a large change, so that a Principle of Least Change might indeed prefer the user's expected change. The other possible reaction is to say that indeed, the desired modification does not satisfy a Least Change Principle in this case, and take this as evidence that there may be cases where following such a principle is undesirable.

Next we look at the most trivial possible examples. For there to be a question about what the best restoration is, there must be inconsistent states, and there must be a choice of how to restore consistency. One moral of this example is that the notion of "size of change" or "amount of disruption" can, even in the simplest case, be context-dependent. Which of two changes appears bigger is interdependent with the way in which changes are represented, which in turn is interdependent with the representation of the models.

Example 4.2 (notQuiteTrivial) *Let $M = \mathbb{B}$ and $N = \mathbb{B} \times \mathbb{B}$ be related by saying that $m \in M$ is consistent with $n = (s, t) \in N$ iff $m = s$. Otherwise, to restore consistency by changing n, we must flip its first component. We have a choice about whether or not also to flip its second component.*

Expressed like that, the example suggests that flipping the second component as well as the first will violate any reasonable Least Change Principle. But this is because the wording has suggested that flipping both components is a larger change than flipping just one. It is possible to imagine situations in which this might not be the case. If n represents the presence or absence of a pebble in each of two pots, and it is possible to create or destroy pebbles, then moving a pebble from one pot to the other might be considered a small change, while creating/destroying a pebble might be considered a large change. In that case, with $m = \top$ and $n = (\bot, \top)$, modifying n to $(\top, \bot)$ might indeed be considered better than modifying it to $(\top, \top)$. This could be captured in a variety of ways, e.g. by a suitable choice of metric on N.

Many bx involve abstracting away information, and intuitively, bx which violate "least change" can arise when a small change in a concrete space results in a large change in an abstract space. Let us illustrate.

Example 4.3 (plusMinus) *Let M be the interval $[-1, +1] \subseteq \mathbb{R}$, and let $N = \{+, -\}$. We say $m \in M$ is consistent with $+$ if $m \geq 0$, and consistent with $-$ if $m \leq 0$. That is, the bx relates a real number with a record of whether it is (weakly) positive or negative.*

Suppose $m < 0$. We have no choice to make about the behaviour of $\overrightarrow{R}(m, +)$: to be correct it must return $-$. On the other hand, $\overleftarrow{R}(m, +)$ could correctly return any non-negative m'. Based on the usual measurement of distance in M, and an intuitive idea of least change, it seems natural to suggest 0 as the choice of m', because it is closer to m than any other correct choice. Dual statements apply for $m > 0$ and $-$.

We will later want to consider two variants on this example.

1. *We change the consistency condition so that $m \in M$ is consistent with $+$ if $m \geq 0$, and consistent with $-$ if $m < 0$ (strictly). This gives a problem in deciding what the result of $\overleftarrow{R}(m, -)$ should be when $m > 0$, because 0 is no longer a correct result, and for any value returned, a "better" one could be found. Observing that this is the result of the model space being non-discrete, we may also consider a second variant:*

2. we replace M by a discrete set, say $\{x/100 : x \in \mathbb{Z}, -100 \leq x \leq 100\}$.

Discreteness of model spaces turns out to be important.

Our next example is adapted from [8], where it is presented in a database context. One moral of this example is that it is not obvious whether to consider it more disruptive to reuse an existing element of a model, or to introduce a new "freely added" element.

Example 4.4 (hegnerInformationOrdering) *Let $M = \mathcal{P}(A \times B \times C)$ for some sets A, B, C, and let $N = \mathcal{P}(A \times B)$, with the consistency relation that $m \in M$ is only consistent with its projection. For example, $m = \{(a_0, b_0, c_0), (a_1, b_1, c_1)\}$ is consistent with $n = \{(a_0, b_0), (a_1, b_1)\}$, but not with $n' = \{(a_0, b_0), (a_1, b_1), (a_2, b_2)\}$. If m is to be altered so as to restore consistency with n', (at least) some triple (a_2, b_2, c) must be added. But what should the value of c be? One may argue that it is better to reuse an already-present element of C, adding (a_2, b_2, c_0) or (a_2, b_2, c_1); or one may argue as Hegner does in [8] that it is better to use a previously unseen element c_2.*

It is reasonably intuitive that just one triple should be added (for example, we should not take advantage of the licence to change m in order to add the legal but unhelpful triple (a_0, b_0, c)); but in certain circumstances, considerations such as those in Example 4.2 may bring even this into doubt.

The same issue arises in our next example, which is adapted from [2]; it also illustrates the current behaviour of QVT-R in the OMG standard semantics. Interestingly in this case the strategy of creating new elements rather than reusing old ones is far less convincing.

Example 4.5 (qvtrPreferringCreationToReuse) *Let M and N be identical model sets, comprising models that contain two kinds of elements* `Parent` *and* `Child`*. Both kinds have a string attribute* ***name****;* `Parent` *elements can also be linked to children which are of type* `Child`*. We represent such models in the obvious way as forests, and notate them using the* ***name****s of elements, e.g. $\{p \to \{c_1, c_2, c_2\}\}$ represents a model containing one element of type* `Parent` *with* `name` *p, having three children of type* `Child` *whose* `name`*s are c_1,c_2,c_2. Notice that we do not forbid two model elements having the same* `name`*.*

The consistency relation between $m \in M$ and $n \in N$ we consider is given by a pair of unidirectional checks. m and n are consistent "in the direction of n" iff for any `Parent` *element in m, say with* ***name*** *t, linked to a child having* ***name*** *c, there exists a(t least one)* `Parent` *element with* ***name*** *t in n, also linked to a child with* ***name*** *c. The check in the direction of m is dual.*

This is expressed in QVT-R as follows (the "enforce" specifications will allow us later to use the same transformation to enforce consistency, but any QVT-R transformation can be run in "check only mode" to check whether given models are consistent):

```
transformation T (m : M ; n : N) {
top relation R {
  s : String;
  firstchild : M::Child;
  secondchild : N::Child;
  enforce domain m me1:Parent {name = s, child = firstchild};
  enforce domain n me2:Parent {name = s, child = secondchild};
  where { S(firstchild,secondchild); }}

relation S {
  s : String;
  enforce domain m me1:Child {name = s};
  enforce domain n me2:Child {name = s};}}
```

Suppose that (for some strings p,c_1,c_2,c_3) we take $m = \{p \to \{c_1, c_2, c_3\}\}$, and for n we take the empty model. When we restore consistency by modifying n, what are reasonable results, from the point of view of least change and considering only the consistency specification?

We might certainly argue that n should be replaced by a copy of m; intuitively, m is a good candidate for the least complex thing that is consistent with m in this case. What the QVT-R specification, and its most faithful implementation, ModelMorf, actually produces is $\{p \to \{c_1\}, p \to \{c_2\}, p \to \{c_3\}\}$.

We refer the reader to [2] for details of the semantics that gives these results, but in brief: QVT-R only changes *model elements when it is forced to do so by "key" constraints. That is, when a certain kind of element is required, and one exists but with the wrong properties, and more than one is not allowed, then QVT-R changes the element's properties. Otherwise, QVT-R modifies a model in two phases. First it adds model elements that are required for consistency to hold. Because it does not change properties of model elements, which include their*

Figure 1: (a) Model space M, (b) model space N for Example 4.7

links (`Parent` to `Child` in our case), it takes an "all or nothing" view of whether an entire valid binding, that is, configuration of model elements that is needed according to a relation, exists. If not, it creates elements for the whole configuration.

So here, the transformation might first discover that it needs a `Parent` named p linked to a `Child` named c_1; since there isn't such a configuration, it creates both elements. Next, it discovers that it needs a `Parent` named p linked to a `Child` named c_2; since such a configuration does not exist, it creates both a new `Parent` and a new `Child`. (We could have written the QVT-R transformation differently, e.g. used a "key" constraint on `Parent`, but this would have other effects that might not be desired.) Similarly for c_3.

Next, with the same m and a further string x, consider $n = \{p \to \{c_1, c_2, x\}\}$. Intuitively, the name of the third `Child` is wrong: it is x and should be changed to c_3. In fact, QVT-R and ModelMorf, using the same transformation as before, actually produce $\{p \to \{c_1, c_2\}, p \to \{c_3\}\}$. As in the previous example, rather than modify an existing `Child`, a whole new binding $\{p \to \{c_3\}\}$ has been created to be the match to the otherwise unmatchable binding involving c_3 in m. In the second phase, the `Child` with `name` x has been deleted, as otherwise the check in the direction of m could not have succeeded.

A final example in [2], omitted here, demonstrates that the question of precisely when elements need to be deleted is problematic. Altogether, modelling modifications as additions and deletions is delicate.

This example is adapted from Example 5.2d of [14].

Example 4.6 (meertensIntersect) *Let $M = \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$, let $N = \mathcal{P}(\mathbb{N})$, and let $m = (s, t)$ be consistent with n iff $s \cap t = n$. Of course $\overrightarrow{R}((s,t), n)$ has no choice; it must ignore n and return $s \cap t$. $\overleftarrow{R}((s,t), n)$ must:*

1. *add to both s and t any element of n that is not already present;*
2. *preserve in both s and t any element of n that is already present;*
3. *delete from* at least one of *s and t any element of their intersection that is not in n.*

As far as correctness goes, it is also at liberty to add any element that is not in n to just one of s, t, provided it is not already present in the other. But intuitively this would be unnecessarily disruptive (in the extreme case where the arguments are already consistent it will violate hippocraticness).

If we take as distance between (s,t) and (s',t') the sum of the sizes of the symmetric differences of the components, we have language in which to express that that behaviour would give a larger change than necessary. Similarly, we justify that in case 3 above, the offending element should be removed from just one set, not both. The choice is arbitrary; Meertens uses the concept of bias to explain how it is resolved by the bx language or programmer. Given one choice of resolution, his calculations produce the result that $\overleftarrow{R}((s,t), n) = (n \cup (s \setminus t), n \cup t)$.

Our next example will enable us to study notions of least change coming from mathematical analysis, and also to illustrate the difference between the "strong" and "weak" least surprise scenarios.

Example 4.7 (weakVsStrongContinuity) *Let $M = \mathbb{R} \cup \{C_M\}$ and $N = (\mathbb{B} \times \mathbb{R}) \cup \{C_N\}$. We use the consistency relation: $R(C_M, C_N)$ and $R(m, (\_, m))$ for all $m \neq C_M$. Figure 1 illustrates. We give M the metric generated by $d_M(C_M, m) = 1 + |m|$ for all $m \in \mathbb{R}$, together with the usual metric $d_M(m, m') = |m - m'|$ on $\mathbb{R}$. Give N the metric generated by the usual metric on both copies of $\mathbb{R}$, together with: $d_N(C_N, (\_, n)) = 1 + |n|$ for all $n \in \mathbb{R}$, and $d_N((\top, n_1), (\bot, n_2)) = 2 + |n_1| + |n_2|$.*

Now, because there is a unique element of M consistent with any given element $n \in N$, there is no choice for $\overleftarrow{R}(m, n)$ given that this must be correct: it must ignore m and return that unique consistent choice. Let us consider the choices we have for the behaviour of $\overrightarrow{R}(m, n)$, and the extent to which each is reasonable from a least change point of view.

- *If m is C_M, there is no choice: we must return C_N to be correct.*
- *If $m = x_m$ and $n = (b, x_n)$ are both drawn from the copies of $\mathbb{R}$, we must return (b', x_m) for some $b' \in \mathbb{B}$. If in fact $x_m = x_n$, hippocraticness tells us we must return exactly n. Otherwise, we could legally return a result which is on the other branch from n, that is, flip the boolean as well as changing the real number. It is easy to argue, though, that to do so violates any reasonable least change principle, since the boolean-flipping choice gives us a change in the model which is larger by our chosen metric, with no obvious prospect for any compensating advantage. Let us suppose we agree not to do so.*
- *The interesting case is $\overrightarrow{R}(x_m, C_N)$ for real x_m. We must return either $(\top, x_m)$ or $(\bot, x_m)$; neither correctness nor hippocraticness place any further restrictions, and when we look at one restoration scenario in isolation, our metric does not help us make the choice, either. We could, for example:*

  1. *pick a boolean value and always use that, e.g. $\overrightarrow{R}(x_m, C_N) = (\top, x_m)$ for all $x_m \in \mathbb{R}$;*
  2. *return $(\top, x_m)$ if $x_m \geq 0$, $(\bot, x_m)$ otherwise; or we could even go for bizarre behaviour such as*
  3. *return $(\top, x_m)$ if x_m is rational, $(\bot, x_m)$ otherwise.*

All of these options for $\overrightarrow{R}(x_m, C_N)$ will turn R into a correct and hippocratic bx. It seems intuitive that these are in decreasing order of merit from a strong least surprise point of view. Imagine that the developers on the M side were not quite sure whether they wanted to put one real number, or another very close to it. The danger that their choice on this matter has determined which branch the N model ends up on, "merely because" the state of the N model at the moment they chose to synchronise "happened" to be C_N, increases as we go down the list of options.

From the point of view of weak least surprise, however, there is no difference between the options, at least for a sufficiently small notion of "small change". For, if $R(m, n)$ holds, and then m is changed to m' by a change that has size greater than 0 but less than 1, it follows that neither m nor n can be the special points C_M and C_N: there are no other models close to C_M, so m has to be one of the real number points, say $x_m \in \mathbb{R}$, m' has to be a nearby real number $x_{m'}$, and from consistency it follows that n must be (b, x_m) for some b. We have already agreed that the result of $\overrightarrow{R}(m', n)$ should be $(b, x_{m'})$.

The key point here is that only in the first option is $\overrightarrow{R}(\_, C_N) : M \to N$ a continuous function in the usual sense of mathematics; in the middle option this function has a discontinuity, while in the final option it is discontinuous everywhere except C_M. This motivates our considerations of continuity in Section 7. Note that not only is $(\{C_M\}, \{C_N\})$ a subspace pair on which any of these variants is continuous (trivially, any pair of consistent states always forms a subspace pair), so is $(M \setminus \{C_M\}, N \setminus \{C_N\})$. This illustrates the potential for a future bx tool to use subspace pairs to warn developers of discontinuous behaviour.

The next example is boring from the point of view of consistency restoration, as consistency is a bijective relation here so there is no choice about how to restore consistency, but it will allow us to demonstrate a technical point later.

Example 4.8 (continuousNotHolderContinuous) *Our model spaces are subsets of real space with the standard metric. Let $M \subseteq \mathbb{R}^2$ comprise the origin, which we label C_M, together with the unit circle centred on the origin, parameterised by θ running over the half-open interval $(0, 1]$. Let $N \subseteq \mathbb{R}$ be $\{0\} \cup [1, +\infty)$. We say that the origin C_M is consistent only with 0, while the point on the unit circle at parameter θ is consistent only with $1/\theta$.*

5 Ordering changes

If we wish to identify changes that are defensibly "least", the most basic thing we can do is to identify the possible changes that could restore consistency, place a partial order on these changes, and insist that the chosen change be minimal. Of course this does not solve anything, and any solution to our problem can be cast in this setting.

Meertens [14] requires structure stronger than this, but weaker than a metric on the model sets. For any model $m \in M$ he assumes given a reflexive and transitive relation on M, notated $x \sqsubseteq_m y$ and read "x is at least as close to m as y is", satisfying the property that $m \sqsubseteq_m x$ for any m, x. If M is a metric space of course we derive this relation from the metric; most of Meertens' examples do actually use a metric, typically size of symmetric difference of sets. He takes it as axiomatic that consistency restoration should give a closest consistent model (according to the preorder), and that it should do so deterministically; much of his paper is devoted to showing

how to calculate systematically a *biased selector* that does this job. Example 4.6 illustrates. Using a similar structure, Macedo et al. [13] address the tricky question of when it is possible to compose bx that satisfy such a least change condition. As might be expected, they have to abandon determinacy (so that the composition can choose "the right path" through the middle model), and impose stringent additional conditions; fundamentally, there is no reason why we would expect bx that satisfy this kind of least change principle to compose.

5.1 Changes as sets of small changes

A preorder on changes is a useful starting point in cases where changes are uncontroversially identified with sets of discrete elements to be added to/deleted from a structure; this is usually taken to be the case for databases. We can then generate a preorder on changes from the inclusion ordering on sets, and this gives a way to prefer changes that do not add or delete elements unnecessarily. Even there, a drawback is that if an element is modified, perhaps very slightly, and we must model this as a deletion of one thing and an addition of something very similar, this change appears bigger than it is. Indeed Example 4.5 is a cautionary tale on how such an approach can produce poor results, where models are structured collections of elements, not just sets.

A similar approach is used by TGGs when used in an incremental change scenario; [11] explains and compares several TGG tools from the point of view of various properties including "least change". In the context of a set of triple graph grammar rules, we suppose given: a derivation of an *integrated triple* $M_S \leftarrow M_C \rightarrow M_T$; that is, a pair of consistent models M_S and M_T together with a correspondence graph M_C; a change Δ_S to M_S. The task is to produce a corresponding change Δ_T to M_T, updating the auxiliary structures appropriately. What the deltas can be is not formally defined in [11] but their property (p7)

> Least change (F4): An incremental update must choose a Δ_T to restore consistency such that there is no subset of Δ_T that would also restore consistency, i.e., the computed Δ_T does not contain redundant modifications.

makes the assumption clear. The issues of handling changes other than as additions plus deletions (mentioned above) and of avoiding creating new elements where old ones could instead of reused (cf Example 4.5 and Example 4.4), are mentioned. Two tools (MoTE and TGG Interpreter) are said to "provide a sufficient means to attain [least change] in practical scenarios" but we are aware of no formal guarantee.

6 Measuring changes

One very natural approach, particularly in the pure state-based setting, is to assume we are given a metric on each model space, and require that the distance between the old, and the chosen new, target models is minimal among all distances between the old target model and any new target model that restores consistency. That is, the effect on their model is as small as it can possibly be, given that consistency must be restored: if this is the case, one argues, it is pointless to insist on more. However, the approach has some limitations, such as failure to compose (essentially because not all triangles are degenerate).

An instance of this approach has been explored and implemented by Macedo and Cunha in [12], although they do not explicitly use the term "metric". They take as given a pair of models and a QVT-R transformation; the QVT-R transformation is used only to specify consistency, the metric-based consistency restoration explored here being used as a drop-in replacement for the standard QVT-R consistency restoration. The models and consistency relation are translated into Alloy, and the tool searches for the nearest consistent model. Their metric is "graph edit distance"; they also briefly considered allowing the user to define their own notion of edits, which amounts to defining their own metric on the space of models.

This approach is very sensitive to the specific metric chosen, and, because it operates after a model has been translated, the graph edit distance used in [12] is not a particularly good match for any user's intuitive idea of distance. There may not be a canonical choice, because different tools for editing models in the same modelling language provide different capabilities. We saw an example of this already in Example 4.2. If my tool provides a simple menu item to make a change that, in your tool, requires many manual steps, is that change small or large? Relative to a specific tool, one can imagine defining a metric by something like "minimal number of clicks and keystrokes to achieve the change", but this is not satisfying when models are likely to be UML models or programs, with hundreds of different available editors. Still, it is reasonable to expect this approach to give more intuitive results than those of the standard QVT-R specification illustrated in Example 4.5.

One still needs a way to resolve non-determinism; it is unlikely that there will be a unique closest consistent model. In [12], the tool offered to the user all consistent models found at the same minimum distance from the

current model. The implementation is very resource intensive and not practical for non-toy cases, and this is probably essential because of the need to consider all models at a given distance from the current one.

We are pessimistic about whether usably efficient algorithms that guarantee to find optimal solutions to the least change problem will ever be available, because Buneman, Khanna and Tan's seminal paper [3] addressing the closely-related *minimal update problem* in databases showed a number of NP-hardness results even in very restricted settings. For example, where S is a database and $Q(S)$ a view of it defined by a query (even a project-join query of constant size involving only two relations), they showed that it is NP-hard to decide, given a tuple $t \in Q(S)$, whether there exists a set $T \subseteq S$ of tuples such that $Q(S \setminus T) = Q(S) \setminus \{t\}$.

Example 4.1 shows, we think, that the model that will be found by this approach will not always be what the user desires in any case. It is possible, though, that by applying exactly the same approach to the witness structure, rather than the model, we might get better behaviour. This would be interesting to investigate.

Formally, for relational bx we may define:

Definition A bx $R : M \leftrightarrow N$ is *metric-least*, with respect to given metrics d_M, d_N on M and N, if for all $m \in M$ and for all $n, n' \in N$, we have

$$R(m, n') \Rightarrow d_N(n, n') \geq d_N(n, \overrightarrow{R}(m, n))$$

and dually.

Note that this is a property which a given bx may or may not satisfy: it does not generally give enough information to *define* deterministic consistency restoration behaviour, because of the possibility that there may be many models at equal distance.

The bx in Example 4.2 will be metric-least or not depending on the chosen metric on N. That in Example 4.3 will be metric-least, with respect to the usual metric on M and with any positive distance between + and − in N. Variant 1 of that example cannot be, however, for the reason given there. Variant 2 is metric-least, regardless of whether 0 is considered consistent with both of + and − or just one. All three variants of Example 4.7 are metric-least.

7 Continuity and other forms of structure preservation

We turn now to codifying reasonable, rather than optimal, behaviour; in a sense we now make the equation "least change = most preservation". The senses in which transformations preserve things have of course been long studied in mathematics (and abstracted in category theory).

The most basic idea, and the most natural way to move on from the metrics-based approach considered in the previous section, is continuity, in the following metric-based formulation. (The other setting in which continuity appears in undergraduate mathematics, topology, we leave as future work.) Informally, a map is continuous (at a source point) if, however close you want to get to your target, you can ensure you get that close by starting within a certain distance of your source. Formally

Definition $f : S \to T$ is continuous at s iff

$$\forall \epsilon > 0 \,.\, \exists \delta > 0 \,.\, \forall s' \,.\, d_S(s, s') < \delta \Rightarrow d_T(f(s), f(s')) < \epsilon$$

We say just "f is continuous" if it is continuous at all s.

Standard results [17] apply: the identity function, and constant functions, are continuous (everywhere), the composition of continuous functions is continuous (at the appropriate points) etc.

To see how to adapt these notions to bx it will help to be more precise. In particular, metric-based continuity of a map is defined at a point: the idea that a map is continuous overall is a derived notion, defined by saying that it is continuous if it is continuous at every point. For us, the points are clearly going to be pairs of models. Supposing that we have metrics d_M, d_N on the model spaces M, N related by a relational bx R:

Definition $\overrightarrow{R}$ is continuous at (m, n) iff

$$\forall \epsilon > 0 \,.\, \exists \delta > 0 \,.\, \forall m' \,.\, d_M(m, m') < \delta \Rightarrow d_N(\overrightarrow{R}(m, n), \overrightarrow{R}(m', n)) < \epsilon$$

This is nothing other than the standard metrics-based continuity of $\overrightarrow{R}(\_, n) : M \to N$ at m. Dually, $\overleftarrow{R}$ is continuous at (m, n) iff $\overleftarrow{R}(m, \_) : N \to M$ is continuous at n.

Definition $\overrightarrow{R}$ is strongly continuous if it is continuous at all (m, n); that is, for every n, $\overrightarrow{R}(\_, n)$ is a continuous function. The definition for $\overleftarrow{R}$ is dual. We say a bx R is strongly continuous if its restorers $\overrightarrow{R}$, $\overleftarrow{R}$ are so.

The terminology "strongly continuous" is justified with respect to our earlier discussion of strong versus weak least surprise, because of the insistence that $\overrightarrow{R}(\_, n)$ is continuous at all m, regardless of whether m and n are consistent.

Definition $\overrightarrow{R}$ is weakly continuous if it is continuous at all *consistent* (m, n); that is, for every n, $\overrightarrow{R}(\_, n)$ is continuous at all points m such that $R(m, n)$ holds. The definition for $\overleftarrow{R}$ is dual. We say a bx R is weakly continuous if its restorers $\overrightarrow{R}$, $\overleftarrow{R}$ are so.

Just as discussed in Section 3, one could consider further variants in which $\overrightarrow{R}(\_, n)$ is required to be continuous at points m where (m, n) is in some other subset of $M \times N$.

Example 4.7 shows that weakly continuous really is weaker than strongly continuous. While all three of the options we considered for $\overrightarrow{R}(m, C_N)$ yield a weakly continuous $\overrightarrow{R}$, only the first gives strong continuity. However, history ignorance – that is, the property that $\overrightarrow{R}(m, \overrightarrow{R}(m', n)) = \overrightarrow{R}(m, n)$, and dually, for all values of m, m', n, generalising PutPut for lenses – makes weak and strong continuity coincide.

Lemma 7.1 *If $R : M \leftrightarrow N$ is history ignorant as well as correct and hippocratic, then $\overrightarrow{R}$ is strongly continuous if and only if it is weakly continuous. Dually this holds for $\overleftarrow{R}$ and hence for R.*

Proof Suppose R is weakly continuous and consider (m, n) not necessarily consistent. We are given ϵ and must find $\delta > 0$ such that

$$\forall m' \,.\, d_M(m, m') < \delta \Rightarrow d_N(\overrightarrow{R}(m, n), \overrightarrow{R}(m', n)) < \epsilon$$

Using the same ϵ, we apply weak continuity at $(m, \overrightarrow{R}(m, n))$ to find δ' such that

$$\forall m' \,.\, d_M(m, m') < \delta' \Rightarrow d_N(\overrightarrow{R}(m, \overrightarrow{R}(m, n)), \overrightarrow{R}(m', \overrightarrow{R}(m, n))) < \epsilon$$

Applying history ignorance, this implies the condition we had to satisfy, so we take $\delta = \delta'$. ∎

This approach has advantages over metric-leastness that we do not have space to explore, but it is worth giving one example. It is straightforward to prove:

Theorem 7.2 *Let $R : M \leftrightarrow N$ and $S : N \leftrightarrow P$ be strongly [rsp. weakly] continuous bx which, as usual, are correct and hippocratic. Suppose further that R is lens-like, i.e., $\overrightarrow{R}$ ignores its second argument; we write $\overrightarrow{R}(m)$. It follows that $\overrightarrow{R}(m)$ is the unique $n \in N$ such that $R(m, n)$. Define the composition $R; S : M \leftrightarrow P$ as usual for lenses: $(R; S)(m, p)$ holds iff there exists $n \in N$ such that $R(m, n)$ and $S(n, p)$; $\overrightarrow{R; S}(m, p) = \overrightarrow{S}(\overrightarrow{R}(m), p)$; $\overleftarrow{R; S}(m, p) = \overleftarrow{R}(m, \overleftarrow{S}(\overrightarrow{R}(m), p))$. Then $R; S$ is also correct, hippocratic and strongly [rsp. weakly] continuous.*

Less positively, continuity is not useful in discrete model spaces, such as those that arise in (non-idealised) model-driven development, because:

Lemma 7.3 *Suppose $m \in M$ is an* isolated point*, in the sense that for some real number $\Delta > 0$ there is no $m' \neq m \in M$ such that $d_M(m, m') < \Delta$. Then with respect to d_M, any $\overrightarrow{R}$ is continuous at (m, n), for every $n \in N$.*

This holds just because for any ϵ one may pick $\delta < \Delta$ and then the continuity condition holds vacuously.

As we would expect, metric-leastness is incomparable with continuity: they offer different kinds of guarantee. All three options in Example 4.7 are metric-least, so metric-leastness does not imply strong continuity. In fact it does not even imply weak continuity; Example 4.3 is metric-least, but not weakly continuous at $(0, +)$. Conversely, Lemma 7.3 shows that even strong continuity does not imply metric-leastness; apply discrete metrics to M and N in Example 4.2, so that by Lemma 7.3 any variant of the bx discussed there is strongly continuous, and pick a variant that is not metric-least by the chosen metric.

7.1 Stronger variants of metric-based continuity

Given that continuity is unsatisfactory because in many of the cases we wish to cover it holds vacuously, a reasonable next step is to consider the standard panoply of strengthened variants of continuity. (Standardly, these definitions *do* imply continuity.) Will any of them be better for our purposes? Let S and T be metric spaces as before.

Definition $f : S \to T$ is uniformly continuous iff

$$\forall \epsilon > 0 \,.\, \exists \delta > 0 \,.\, \forall s, s' \,.\, d_S(s, s') < \delta \Rightarrow d_T(f(s), f(s')) < \epsilon$$

That is, in contrast to the standard continuity definition, δ depends only on ϵ, not on s.

This, adapted to bx, will obviously also be vacuous on discrete model spaces, so let us not pursue it.

Definition Given non-negative real constants C, α, we say $f : S \to T$ is Hölder continuous (with respect to C, α) at s iff

$$\forall s' \,.\, d_T(f(s), f(s')) \leq C d_S(s, s')^\alpha$$

We say that f is Hölder continuous if it is so at all s. The special case where $\alpha = 1$ is known as Lipschitz continuity.

Note that, to be congruent with our other definitions and for ease of adaptation, we have defined Hölder continuity first at a point, and then of a function. Since the definition is symmetric in s and s', it is not often presented that way. Adapting to bx as before by considering the Hölder continuity of $\overrightarrow{R}(\_, n) : M \to N$ at m, we get

Definition $\overrightarrow{R}$ is Hölder continuous (with respect to C, α) at (m, n) iff

$$\forall m' \,.\, d_N(\overrightarrow{R}(m, n), \overrightarrow{R}(m', n)) \leq C d_M(m, m')^\alpha$$

Then as before, we may say that R is strongly (C, α)-Hölder continuous if it is so at all (m, n), weakly (C, α)-Hölder continuous if it is so at consistent (m, n), and we may consider intermediate notions if we wish.

The fact that the adaptation to bx is symmetric in m and m' raises the question of whether strong Hölder continuity is actually stronger than weak Hölder continuity, however. In fact Example 4.7 was designed to demonstrate this: for example, variant 2 is easily seen to be weakly (1,1)-Hölder continuous, but is not strongly (1,1)-Hölder continuous because we can pick $n = C_N$ and m, m' to be real numbers which are arbitrarily close but on opposite sides of 0.

We get again the analogue of Lemma 7.1, by the same argument.

Lemma 7.4 *If $R : M \leftrightarrow N$ is history ignorant as well as correct and hippocratic, then R is strongly (C, α)-Hölder continuous if and only if it is weakly (C, α)-Hölder continuous.*

The next interesting question is whether Hölder continuity might avoid the problem we noted for continuity, viz. that it is trivially satisfied at isolated points. The answer is that it does, as Example 4.8 (which, recall, actually had no choice about how to restore consistency) shows: although $\overrightarrow{R}$ is continuous at every (m, n), it is not (C, α)-Hölder continuous at (C_M, n) for *any* $n \in N$ because, while the distance between C_M and any other m' is 1, the distance between $\overrightarrow{R}(C_M, n) = 0$ and $\overrightarrow{R}(m', n)$ can be made arbitrarily large by judicious choice of m'.

Thus it is possible that Hölder continuity might turn out to be a useful least change principle, and we leave this as an open research question; we remark, though, that continuity is already a strong condition, Hölder continuity much stronger, and it seems more likely that this will be a "nice if you can get it" property rather than one it is reasonable to insist on. Still, it might be interesting to consider, for example, subspace pairs on which it holds, and the possibility that a tool might indicate when a user leaves such a subspace pair.

Future work might consider e.g. locally Lipschitz functions, or in a sufficiently special space, continuously differentiable functions, etc.

8 Category theory

As previously discussed in Section 5, various authors have investigated least change in a partially (or even pre-) ordered setting. A natural generalisation of such work is to move from posets to categories. In particular, a number of people (notably Diskin *et al.* [5], Johnson, Rosebrugh *et al.* [10, 9, among others]) have considered generalisations of very well-behaved (a)symmetric lenses from the category of Sets to more general settings, notably Cat itself, the category of small categories.

The basic idea underlying these approaches is to go beyond the basic set-theoretic (state-based, whole-update) approach of lenses in order to incorporate additional information about the updates themselves, modelled as arrows. Rather than consider models, database states, as *elements* of an unstructured *set* (corresponding to a *discrete* category), they are taken as objects of a category $\mathbb{S}$. Arrows $\gamma : S \longrightarrow S'$ correspond to *updates*, from old state S to new state S'. Arrows from a given S carry a natural preorder structure induced by post-composition, generalising the order induced by multiplication in a monoid:

$$\gamma : S \longrightarrow S' \leq \gamma' : S \longrightarrow S'' \quad \text{iff} \quad \exists \delta : S' \longrightarrow S'' . \gamma' = \gamma ; \delta$$

Johnson, Rosebrugh and Wood introduced the idea of a **c-lens** [10] as the appropriate categorical generalisation of very-well-behaved asymmetric lens: given by a functor $G : \mathbb{S} \longrightarrow \mathbb{V}$ specifying a *view* of $\mathbb{S}$, together with data defining the analogues of Put, satisfying appropriate analogues of the GetPut, PutGet and PutPut laws (we omit

the details, which are spelt out very clearly, if compactly, in their subsequent paper [9]). They make explicit the connection with, and generalisation of, Hegner's earlier work characterising least-change update strategies on databases [7]; in the categorical setting, database instances are models of a sketch defining an underlying database schema or entity-relationship model.

The crucial detail is that the 'Put' functor then operates not on pairs of states and views alone (that is, objects of $\mathbb{S} \times \mathbb{V}$), but on updates from the image of some state S under G to a new view V, that is, on pairs consisting of an $\mathbb{S}$-state S and a $\mathbb{V}$-arrow $\alpha : GS \longrightarrow V$, returning a new $\mathbb{S}$-arrow $\gamma : S \longrightarrow S'$ such that $G\gamma = \alpha$. In other words, Put not only returns a new updated state S' on the basis of a updated view V, but also an update from S to S' that is correlated with the view update α. Notice that, because we have an update to a source S correlated with an update to a view GS which is *consistent* with the source, we are in the *weak* least surprise setting – but since very-well-behavedness in their framework is history-ignorance, the notions of weak and strong least surprise coincide.

They then show that the laws for Put establish that such an update γ is in fact *least* (indeed, *unique*) up to the $\leq$ ordering, namely as an *op-cartesian* lifting of α. Thus very-well-behaved asymmetric c-lenses do enjoy a Principle of Least Change. Extending this analysis, which applies to *insert* updates, with *deletes* being considered dually via a *cartesian* lifting condition on arrows $\alpha : V \longrightarrow GS$, to the symmetric case is the object of ongoing study in terms of spans of c-lenses.

Johnson and Roseburgh further showed that Diskin *et al.*'s *delta lenses* [5] generalise the c-lens definition; in particular, every c-lens gives rise to an associated d-lens. However, such d-lenses do *not* enjoy the unique arrow-lifting property, so there is some outstanding issue about the generality of the d-lens definition. In subsequent work [6], Diskin *et al.* have given a definition of symmetric d-lenses. It remains to be seen what least-change properties such structures might enjoy, on their own, or by reference to spans of c-lenses.

9 Conclusions and future work

The vision of bx in MDD is that developers should be able to work on essentially arbitrary models, which capture just what is relevant to them, supported by languages and tools which make it straightforward to define consistency and restore consistency between their model and others being used elsewhere. Clearly, if anything like this is to be achieved, there is vastly more work to be done on all fronts. Although there are some islands of impressive theoretical results (e.g. in category theory) and some pragmatically useful tools (e.g. based on TGGs), the theory currently works only in very idealised settings and the tools offer inadequate guarantees while still lacking flexibility. For software development, this situation limits productivity. For researchers it is an adventure playground.

We understand enough already to be sure that we will never achieve behaviour guarantees as strong as we would ideally like within the flexible tools we need. But the limits of what can be achieved are unknown. How far can we go, for example, by identifying "good" parts of the model spaces, where guarantees can be offered, and developing tools that warn their users when danger is near, so that they can spend their attention where it is most needed? If we need information from users to define domain-specific metrics, how can we elicit this information without unacceptably burdening users? Can we do better by focusing on *reasonable* behaviour than on *optimal* behaviour? It is noteworthy that, despite the name, HCI experts following the Principle of Least Surprise (or Astonishment) there are not really making an optimality claim so much as a reasonableness one: interface users might not know precisely what to expect, but when they see what happens, they should not be surprised. We think this is a good guide.

We have been pointing out, through the paper, areas we think need work (or play). Let us now mention some others that we have not touched on here. We have not mentioned topology, although we have touched on both metric spaces (a specialisation) and category theory (a generalisation). Perhaps the language of topology, or even algebraic topology, might help us to make progress. Even more speculatively, as type theory, especially dependent type theory, is a language we work in elsewhere, it is natural to wonder whether at some point spatial aspects of types such as for example homotopy type theory will have a role.

We have not attempted to analyse which properties any of the existing formalisms provide or could provide. Do any of the many existing bx formalisms that we have not mentioned, each thoughtfully designed in an attempt to "do the right thing", satisfy any of the properties discussed here – and if not, why not? Under what circumstances do only global optima exist, so that guaranteeing reasonable behaviour would automatically guarantee optimal behaviour? Would identifying such circumstances help to resolve the tension between wanting optimality and wanting composition? Is there any mileage in applying the kind of guarantees of reasonable

behaviour considered here to partial bx in the sense of [16], which do not necessarily restore consistency but, in an appropriate sense, at least improve it? Could someone make use of insights from Lagrangian and Hamiltonian mechanics? Or from simulated annealing?

Nailing our colours to the mast, we think: change to witness structures is important; pursuing reasonable behaviour will be more fruitful than pursuing optimal behaviour; identifying "good" subspace pairs will help tools in practice; and weak least surprise is not enough. But there is plenty of room for other opinions.

Acknowledgements

We thank the anonymous reviewers, Faris Abou-Saleh, and the bx community for helpful comments and discussions. The work was funded by EPSRC (EP/K020218/1, EP/K020919/1).

References

[1] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *Proceedings of MPC*, number 9129 in LNCS, 2015. To appear.

[2] Julian C. Bradfield and Perdita Stevens. Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In *Proceedings of FASE*, volume 7212 of *LNCS*, pages 194–208. Springer, 2012.

[3] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of PODS*, pages 150–158. ACM, 2002.

[4] James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a repository of bx examples. In K. Selçuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Workshops of EDBT/ICDT*, volume 1133 of *CEUR Workshop Proceedings*, pages 87–91. CEUR-WS.org, 2014.

[5] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.

[6] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *Proceedings of MODELS*, pages 304–318, 2011.

[7] Stephen J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Artif. Intell.*, 40(1-2):63–125, 2004.

[8] Stephen J. Hegner. Information-based distance measures and the canonical reflection of view updates. *Ann. Math. Artif. Intell.*, 63(3-4):317–355, 2011.

[9] Michael Johnson and Robert D. Rosebrugh. Delta lenses and opfibrations. *ECEASST*, 57, 2013.

[10] Michael Johnson, Robert D. Rosebrugh, and Richard J. Wood. Lenses, fibrations and universal translations. *Mathematical Structures in Computer Science*, 22:25–42, 2 2012.

[11] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. *ECEASST*, 67, 2014.

[12] Nuno Macedo and Alcino Cunha. Implementing QVT-R bidirectional model transformations using alloy. In Vittorio Cortellessa and Dániel Varró, editors, *Proceedings of FASE*, volume 7793 of *LNCS*, pages 297–311. Springer, 2013.

[13] Nuno Macedo, Hugo Pacheco, Alcino Cunha, and José Nuno Oliveira. Composing least-change lenses. *ECEASST*, 57, 2013.

[14] Lambert Meertens. Designing constraint maintainers for user interaction. Unpublished manuscript, available from http://www.kestrel.edu/home/people/meertens/, June 1998.

[15] Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Journal of Software and Systems Modeling (SoSyM)*, 12(1):175–199, 2013. Published online, 16 March 2011.

[16] Perdita Stevens. Bidirectionally tolerating inconsistency: Partial transformations. In Stefania Gnesi and Arend Rensink, editors, *Proceedings of FASE*, volume 8411 of *LNCS*, pages 32–46. Springer, 2014.

[17] W A Sutherland. *Introduction to metric and topological spaces*. Oxford University Press, 1975.

A Systematic Approach and Guidelines to Developing a Triple Graph Grammar

Anthony Anjorin
Chalmers | University of Gothenburg
anjorin@chalmers.se

Erhan Leblebici, Roland Kluge, Andy Schürr
Technische Universität Darmstadt
{firstname.lastname}@es.tu-darmstadt.de

Perdita Stevens
University of Edinburgh
perdita.stevens@ed.ac.uk

Abstract

Engineering processes are often inherently concurrent, involving multiple stakeholders working in parallel, each with their own tools and artefacts. Ensuring and restoring the consistency of such artefacts is a crucial task, which can be appropriately addressed with a bidirectional transformation (*bx*) language. Although there exist numerous *bx* languages, often with corresponding tool support, it is still a substantial challenge to learn how to actually *use* such *bx* languages. Triple Graph Grammars (TGGs) are a fairly established *bx* language for which multiple and actively developed tools exist. Learning how to master TGGs is, however, currently a frustrating undertaking: a typical paper on TGGs dutifully explains the basic "rules of the game" in a few lines, then goes on to present the latest groundbreaking and advanced results. There do exist tutorials and handbooks for TGG tools but these are mainly focussed on how to use a particular tool (screenshots, tool workflow), often presenting exemplary TGGs but certainly not how to derive them systematically. Based on 20 years of experience working with and, more importantly, explaining *how* to work with TGGs, we present in this paper a systematic approach and guidelines to developing a TGG from a clear, but unformalised understanding of a *bx*.

1 Introduction and Motivation

Formalizing and maintaining the consistency of multiple artefacts is a fundamental task that is relevant in numerous domains [6]. Bidirectional transformation (*bx*) languages address this challenge by supporting bidirectional change propagation with a clear and precise semantics, based on a central notion of consistency specified with the *bx* language. Although numerous *bx* approaches exist [19], often with corresponding tool support, mastering a *bx* language is still a daunting task. Even once the user has a clear, though unformalised, understanding of what the *bx* should do, embodying this understanding in a *bx* is non-trivial. Formal papers do not address this issue, and neither, usually, do tool-centric handbooks.

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

Triple Graph Grammars (TGGs) [18] are a prominent example of a *bx* language for which this observation holds: a typical paper on TGGs dutifully explains how TGGs "work" in a few lines, then conjures up a complete and perfect TGG specification for the running example out of thin air. This is not how things work in practice; going from a clear, but unformalised understanding of consistency to a formal TGG specification is difficult, especially for developers who do not already have ample practice with rule-based, declarative, (graph) pattern-based languages. Existing tutorials, handbooks, and introductory papers on TGGs do not alleviate this situation as they are either tool-specific, focusing on how to use a certain TGG tool, or present basic concepts without providing a systematic approach to how TGG rules can be iteratively engineered.

To the best of our knowledge, the only existing work in this direction is from Kindler and Wagner [13] and later, treated in some more detail in Wagner's PhD thesis [22]. Similarly to [21] for model transformation, Kindler and Wagner propose an algorithm that "synthesizes" TGG rules from a set of consistent examples provided by the user. Although this can be very helpful in scenarios where rather simple but numerous TGG rules are required, we have observed that a typical TGG involves a careful design process with direct consequences for the resulting behaviour of derived model synchronizers. Indeed, Kindler and Wagner mention that synthesized TGGs probably have to be adjusted, extended and finalized manually, but do not provide adequate guidance of how this can be done systematically.

Based on 20 years' experience working together with industrial partners and students, learning how to understand and use TGG specifications, our contribution in this paper is, therefore, to provide a systematic approach to creating and extending TGGs. Our aim with the resulting step-by-step process, is to substantially lower the initial hurdle of concretely working with TGGs, especially for other users and researchers in the *bx* community.

The rest of the paper is organized as follows: Sect. 2 reviews the preliminaries of TGGs and provides our running example that is available as `Ecore2Html` from the *bx* example repository [5], and as a plug-and-play virtual machine hosted on *Share*.[1] Sect. 3 introduces an iterative approach and a set of guidelines for engineering a TGG, constituting our main contribution in this paper. Sect. 4 complements this by providing an intuitive understanding of TGG-based synchronization algorithms. Sect. 5 compares our work to other related contributions. Sect. 6 summarizes and gives a brief outlook on future tasks.

2 Running Example and Preliminaries

The running example used in the rest of this paper is a bidirectional transformation between class diagrams and a corresponding HTML documentation. Consider, for instance, a software development project for implementing a TGG-based tool. The project consists of two groups of developers: (i) core developers who are responsible for establishing and maintaining the main data structures and API provided by the tool, and (ii) other developers who mainly use the API and work with the tool (functioning as beta testers for the project). The latter group does not *define* the data structures involved, but is probably in a better position to *document* all packages, classes, and methods. It thus makes sense to maintain two types of artefacts for the two groups of stakeholders: (1) class diagrams, maintained by core developers, and (2) HTML documents, maintained by API users in, e.g., a wiki-like manner. The class diagram to the left of Fig. 1 depicts the main data structures of a TGG tool, with a corresponding folder structure containing HTML files to the right.

The outermost package `TGGLanguage` corresponds to the top-most folder `TGGLanguage`, and contains subpackages and classes, which correspond to subfolders and HTML files, respectively. `TripleGraphGrammar`s consist of `TGGRule`s, which are in essence graph patterns (referred to as `StoryPattern`s) consisting of variables that stand for objects (`ObjectVariable`s) and links (`LinkVariable`s) in a model. This general concept of a graph pattern is extended for TGGs by attaching a `Domain` to each concept. Each `Domain` references a `Metamodel`, a wrapper for the supported metamodelling standard.[2] The classes `StoryPattern`, `LinkVariable`, `ObjectVariable`, and `EPackage` are imported from other class diagrams (greyed out) and are not part of the documentation of `TGGLanguage`. The corresponding documentation model is much simpler: it consists of a folder structure mirroring the package structure in the class diagram, and an HTML file for each class. Note that packages are also documented in corresponding files, which start with "__" to distinguish them from class documentation files (e.g., `__TGGLanguage.html`). Inheritance and references are irrelevant for the documentation model, but all methods of each class (excluding inherited methods) are to be documented as a table with two columns (name of the method and documentation) in the corresponding class documentation file.

[1] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=LookupImage&bNameSearch=eMoflon`

[2] In this case `EPackage` for *Ecore*, the *de facto* standard of the Eclipse Modeling Framework (EMF).

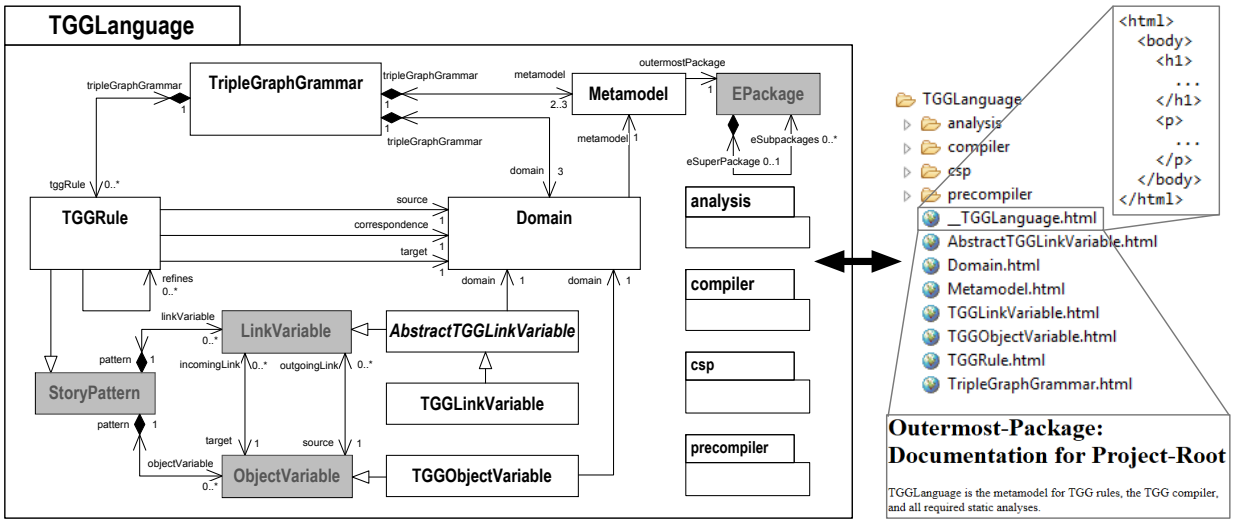


Figure 1: A class diagram and its corresponding HTML documentation

The following points are noteworthy: (1) Information loss is incurred when transforming class diagrams to documentation models (e.g., inheritance relations and references) *and* when transforming documentation models to class diagrams (all entered documentation is lost!). This is the main reason why the change propagation strategy required for this example *must* take the old version of the respective output artefact into account. (2) Not all possible changes make sense: it is arguable whether API users should be able to rename elements in the documentation and propagate such changes to the corresponding class diagrams. Even more arguable are changes such as adding new elements in the documentation that do not yet exist in the class diagram. Such changes may be interpreted as feature requests, but are probably not primary use cases.

Models, Metamodels, Deltas, and Triple Graph Grammars: We assume a very basic understanding of TGGs and of Model-Driven Engineering (MDE) concepts, and refer readers having a hard time understanding the following to, e.g., the eMoflon handbook[3] for a gentle, tutorial-like introduction. Readers more interested in a formal and detailed introduction to TGGs are referred to, e.g., [1].

In the following, we introduce core terms and notation as they are to be understood (informally) in this paper. Based on this understanding, we then propose a systematic, iterative TGG *development process* in Sect. 3, together with a set of *guidelines* or best practices. This process is then demonstrated by applying it to develop a TGG for our running example.

A *model* is an abstraction (a simplification) of something else, chosen to be suitable for a certain task. In an MDE context, *metamodels*, essentially simplified UML class diagrams, are used to define languages of models.

Given two languages of models and their respective metamodels, say a *source* and a *target* language, a *consistency relation* over source and target models is a set of pairs of a source and a target model, which are to be seen as being consistent with each other.

Given a consistent pair of source and target models, a change applied to the source model is referred to as a *source delta*, and a change to the target model is called a *target delta*. As models are graph-like structures consisting of attributed nodes and edges, we may decide that *atomic deltas* are one of: element (node/edge) creation, element deletion, and attribute changes. Atomic deltas can be composed to yield a *composite delta*. A composite delta that only involves element creation is called a *creating delta*. A source/target delta that does nothing is called an *idle source/target delta*, respectively.

Given a consistent pair of source and target models, and a source delta, the task of computing a corresponding target delta that restores consistency by changing the target model appropriately is referred to as *forward model synchronization*, or just model synchronization. This applies analogously to target deltas and computed source deltas, i.e., *backward model synchronization*. The more general task of restoring consistency given both a source and a target delta is referred to as *model integration*, which is outside the scope of this paper.

A *Triple Graph Grammar* (TGG) is a finite set of *rules* that each pair a creating source/target delta with either a corresponding creating target/source delta or with an idle target/source delta, respectively.

[3]Available from `www.emoflon.org`

Each TGG rule states that applying the specified pair of deltas together will extend a given pair of source and target models consistently. To keep track of correspondences between consistent source and target models on an element-to-element basis, a third *correspondence model* can be maintained consisting of explicit *correspondence elements* connecting certain source and target elements with each other. These correspondence elements are typed with a correspondence metamodel, which can be chosen as required. In general, a TGG can thus be used to generate a language of *triples*, consisting of connected source, correspondence, and target models. A triple is denoted as $G_S \leftarrow G_C \rightarrow G_T$ (G for *graph*). A TGG induces a consistency relation as follows: A pair of source and target models (G_S, G_T) is *consistent* if there exists a triple $G_S \leftarrow G_C \rightarrow G_T$ that can be generated by applying a sequence of deltas, as specified by the rules of the TGG.

A *TGG tool* is able to perform model synchronization using a given TGG as the specification of consistency. A TGG tool is *correct* if it only produces results that are consistent according to the given TGG used to govern the synchronization process [18]. As correctness does not in any way imply that information loss should be avoided, one TGG tool is said to be more *incremental* than another if it handles (avoids) information loss better [1]. A TGG tool is said to *scale* if the runtime for synchronization depends polynomially (and not exponentially) on model and delta size, and is *efficient* if the runtime for synchronization only depends on delta size (and no longer on model size) [1].

Example: To explain these basics further, Fig. 2 depicts the source and target metamodels chosen for the running example. The source metamodel (to the left) is *Ecore*, a well-known standard for simple class diagrams. Class diagrams basically consist of packages (`EPackage`s) containing subpackages, and classes (`EClass`es) with methods (`EOperation`s). The target metamodel (to the right) is *MocaTree*, a generic tree-like target metamodel providing concepts to represent a hierarchy of folders with files, where each file contains a tree consisting of `Node`s. Objects of type `Text` but not of type `Node` are used to represent leaves in the tree that cannot have children. In this manner, it is possible to represent arbitrary (X)HTML trees. The correspondence metamodel consists of types connecting source and target elements as required for the rules and is omitted here.

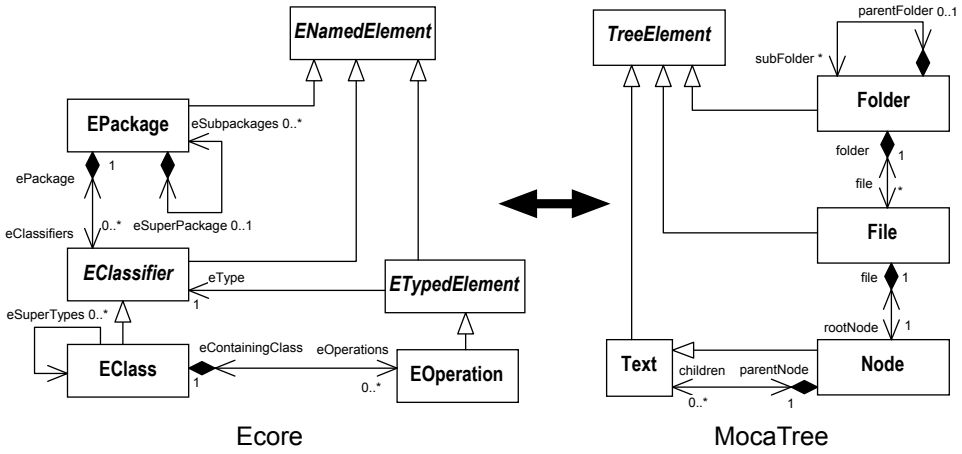


Figure 2: Source and target metamodels for the running example

To introduce the notation and language features used in this paper to specify TGG rules, let us consider two TGG rules for handling (sub)packages and their corresponding documentation. The TGG rule for handling *root* packages is depicted to the left of Fig. 3. The source and target deltas paired by this rule are: (i) creating a new `EPackage` in the source model, and (ii) creating a `Folder` with a single HTML `File`. The basic structure of the HTML file, consisting of a header and a paragraph where a description of the package can be entered, is also created in the same target delta. The package and folder are connected with a correspondence node of type `EPackageToFolder`. In standard TGG visual notation, all elements *created* in a rule are depicted in green with an additional `"++"` markup for black and white printouts. Correspondence nodes are additionally depicted as hexagons to clearly differentiate them from source and target elements.

A set of *attribute constraints* specifies the relationship between attribute values of the elements in a TGG rule. All TGG tools provide an extra, typically simple textual language for expressing such attribute constraints. In this case, the `eq` constraint expresses that `ePackage` has the same name as `docuFolder`. To express that the name of the `htmlFile` should start with `"__"`, end with `".html"`, and contain the name of `ePackage`, two attribute constraints `addSuffix` and `addPrefix` are used, where `withSuffix` is a temporary variable.

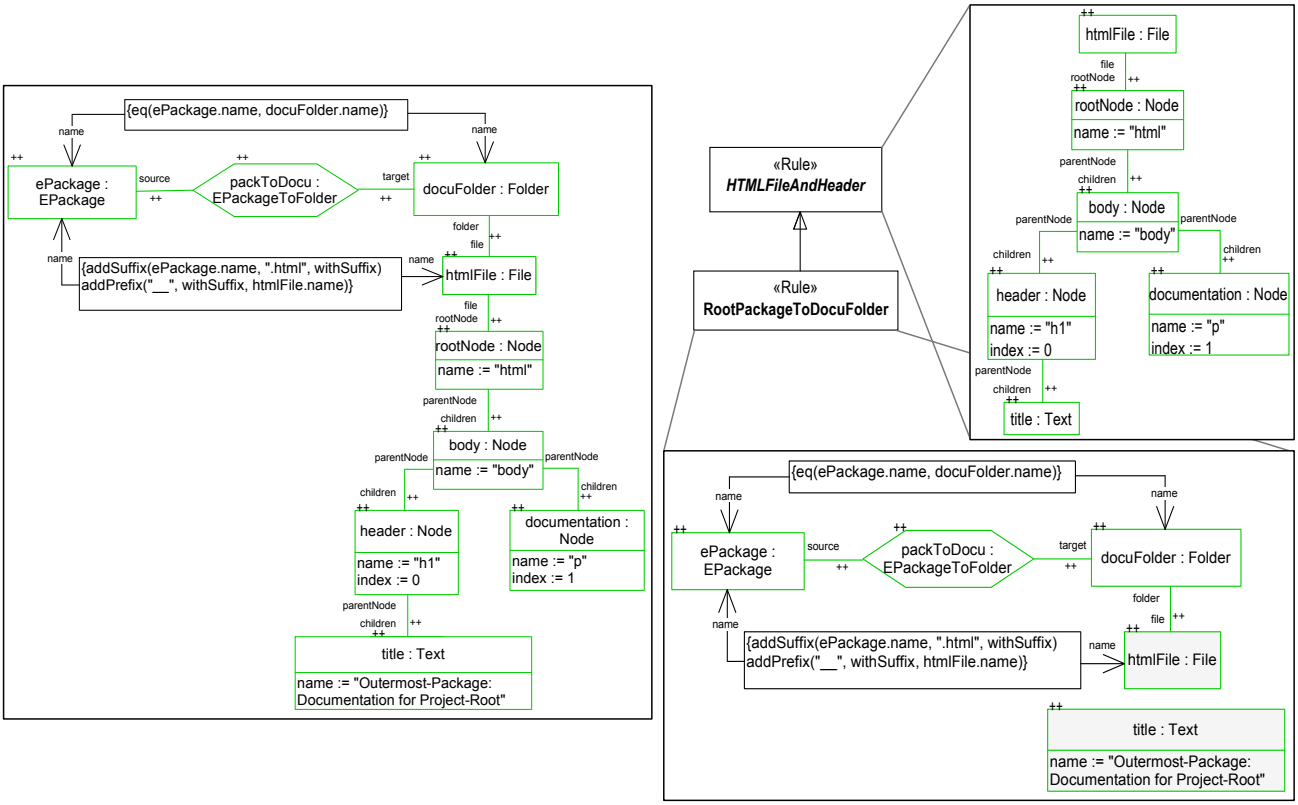


Figure 3: Handling root packages: without (left) and with (right) rule refinement

For cases where an attribute is simply assigned a constant, this can be "inlined" in the respective node, e.g., `name := "html"` in `rootNode`. If we consider Fig. 1 again, we can now compare the creating deltas in the TGG rule to the package `TGGLanguage`, the folder `TGGLanguage` and the HTML file `__TGGLanguage.html`, and see that all attribute constraints are fulfilled.

It is often useful to extract parts of a rule into a *basis rule*, so that these parts can be reused in other *subrules*. Apart from possibly enabling reuse, readability can also be increased by decomposing a rule into modular fragments that each handle a well-defined concern. Many TGG tools support some form of *rule refinement* [3, 8, 14] as depicted to the right of Fig. 3: A new TGG rule `HTMLFileAndHeader`, creating the basic structure of an HTML file, has been extracted and is now *refined* (denoted by an arrow) by the subrule `RootPackageToDocuFolder` to yield the same rule as depicted to the left of Fig. 3. As we do not want to allow creating basic HTML files on their own, `HTMLFileAndHeader` is declared to be *abstract* (denoted by displaying the rule's name in italics). This means that it is solely used for modularization and not for synchronization. The exact details of rule refinement are out-of-scope for this paper, but in most cases (as here), it is a merge of the basis with the refining rule, where elements in the refining rule override elements with the same name in the basis rule (this is the case for `htmlFile` and `title`). Such overriding elements are shaded light grey in subrules to improve readability. TGGs with rule refinements are flattened to normal TGGs. Abstract rules are used in the process, but are not included in the final TGG. Non-abstract rules are called *concrete* rules.

Fig. 4 depicts a further TGG rule `SubPackageToDocuFolder` for handling subpackages (`EPackage`s with a super package). `SubPackageToDocuFolder` refines `RootPackageToDocuFolder` and additionally places the created package into a super package, and the corresponding subfolder into a parent folder (determined using the correspondence node `superToFolder`). Finally, the title is adjusted for subpackages by overriding it in `SubPackageToDocuFolder`. This rule shows how *context* can be demanded; the *context elements* `superPackage`, `superToFolder`, `superFolder`, and connecting `source` and `target` links are depicted in black and must be present (created by some other rule) before the rule can be applied. In this sense, the context elements form a *precondition* for the application of the rule.

Our current TGG consists of two concrete rules and one abstract rule, and can be used to generate consistent package hierarchies and corresponding folder structures with HTML files for documenting the packages.

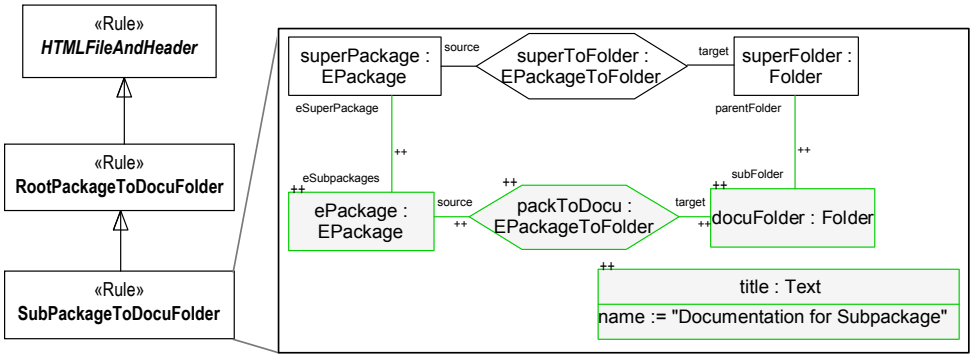


Figure 4: Handling subpackages and their documentation

In the following section, we shall take a look at how to systematically develop TGG rules in a step-by-step process, discussing various *kinds* of TGG rules and how design choices affect derived TGG-based synchronizers.

3 An Iterative Process and Guidelines for Developing a TGG

To introduce basic concepts we have already specified a TGG to handle package structures and their HTML documentation. This was done in a relatively ad-hoc fashion, choosing source and target metamodels, deciding to start with handling packages, and specifying the required TGG rules without consciously applying any systematic process. Although this might work fine for simple cases or for seasoned TGG experts, we propose the following process depicted in Fig. 5, which consists of four steps outlined in the following sections. In each step, we give guidelines that represent current best practice and provide guidance when making design decisions.

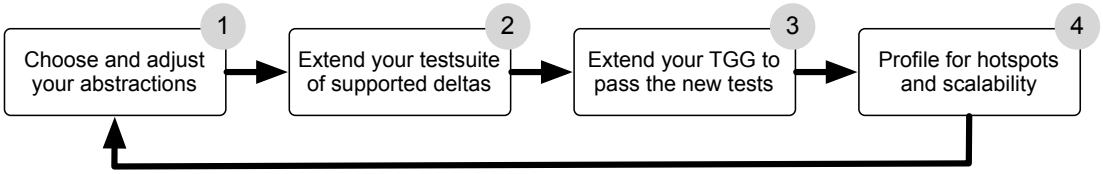


Figure 5: Developing a TGG with an iterative, test-driven development process

3.1 Choose and adjust your abstractions (metamodels)

In practice, the final artefacts to be synchronized are typically fixed for a certain application scenario, e.g., XML files, a certain textual format, a tool's API, or programs in some programming language. As TGG tools do not usually operate directly on these final artefacts, a parser/unparser component, referred to as an *adapter* in the following, is required to produce a graph-like abstraction of the artefacts. The choice of source/target metamodels thus becomes a degree of freedom with a direct impact on the complexity of the required adapter.

Guideline 1 (Adapter complexity vs. rule complexity)**.** *Choosing very high-level source and target metamodels might lead to simple, elegant TGG rules but also requires complex adapters in form of (un)parsers or some other import/export code. As this shifts complexity away from the TGG, strive to keep such pre/post-processing adapter logic to a minimum. Decomposing the synchronization into multiple steps by introducing an intermediate metamodel and developing two TGGs instead of one can be a viable alternative.*

Example: For our running example, the source artefacts are XMI files representing class diagrams, while target artefacts are XHTML files. As the class diagrams already conform to Ecore, choosing Ecore as the source metamodel means that the standard EMF XMI reader and writer can be used. To handle XHTML files, a simple and generic XML adapter is used that produces instances of a basic, tree-like metamodel `MocaTree`, as depicted in Fig. 2. Although these choices reduce the required adapter logic to a minimum (G1), our TGG rules are rather verbose, especially in the target domain (cf. Fig. 3). We addressed this with rule refinement (any form of modularity concept could be beneficial in this case if supported by the chosen TGG tool), but could also have chosen a richer target metamodel and shifted most of the complexity to a problem-specific XHTML parser and unparser instead.

3.2 Extend your test suite of supported deltas

Although it is generally accepted best practice in software development to work iteratively and to apply regression testing, beginners still tend to specify multiple rules or even a complete TGG without testing. This is particularly problematic because the understanding of the consistency relation often grows and changes *while* specifying a TGG. The following guideline highlights that any non-trivial TGG should be supported by a test suite.

Guideline 2 (Take an iterative, test-driven approach)**.** *When specifying a TGG, take a test-driven approach to prevent regression as the rules are adjusted and extended. Run your test suite after every change to your TGG.*

In this context, a source *test case* consists of: (i) a consistent triple (possibly empty), (ii) a source delta to be applied to the triple, and (iii) a new target model representing the expected result of forward synchronizing the source delta. Target test cases are defined analogously. Although all important deltas for an application scenario should eventually be tested, it makes sense to focus first on creating deltas as these can be almost directly translated into TGG rules.

Guideline 3 (Think in terms of creating source and target deltas)**.** *Think primarily in terms of consistent pairs of creating source and target deltas and derive corresponding test cases. This simplifies the transition to a TGG.*

The following two guidelines propose to handle "easy" cases first before going into details. Concerning creating deltas, "easy" translates to "not dependent on context".

Guideline 4 (Start with context-free creating deltas)**.** *Start with context-free pairs of creating deltas as these are usually simpler. A context-free delta can always be propagated in the same way, independent of what the elements it creates will be later connected with.*

Although graphs do not have a "top" or "bottom" in general, in many cases models do have an underlying containment tree. Top-down thus means container before contents. As containers typically do not depend on their contents, we get the following guideline as a corollary of Guideline 4:

Guideline 5 (Prefer top-down to bottom-up)**.** *If possible, start top-down with roots/containers of the source and target metamodels as containers typically do not depend on their contents.*

Example: Applying these guidelines to our running example, we chose to handle package hierarchies in a first iteration (G2). We started thinking about creating a root package in the class diagram as a source delta, whose corresponding target delta is creating a folder containing a single HTML file named "__" + `ePackage.name`, which is to contain the documentation for the root package (G3). This is always the case irrespective of what the root package/folder later contains and is thus a context-free pair of creating deltas (G4). We took a top-down approach (G5) handling root packages before subpackages (cf. Fig. 3 and Fig. 4).

3.3 Extend your TGG to pass the new tests

After discussing with domain experts and collecting test cases addressing a certain aspect, e.g., (sub)packages and folders, the next step is to specify new, or adjust existing, TGG rules to pass these tests. To accomplish this, it is crucial to understand that there are only a few basic *kinds* of TGG rules as depicted schematically in Fig. 6. The names are chosen to give a geographical intuition, where green and black clouds/arrows denote created and context elements, respectively:

Islands are context-free rules that do not demand any context. An island may either be idle, creating either source or target elements, or it may create both, source *and* target elements. After applying such a rule, an isolated "island" of elements is created.

Extensions require a context island and then *extend* it by creating and directly attaching new elements to it.

Bridges connect two context islands by creating new elements to form a *bridge* between the islands. Bridges can of course be generalized to connect multiple islands, but remain fundamentally the same. Bridges connecting more than two islands are rare in practice, probably because they become increasingly complex to comprehend.

Ignore rules (not depicted) are TGG rules that do not create any elements in either the source or target domain. Such rules state explicitly that applying a certain creating delta in one domain has no effect on the other domain.

Based on these basic kinds of TGG rules, we propose the following guidelines.

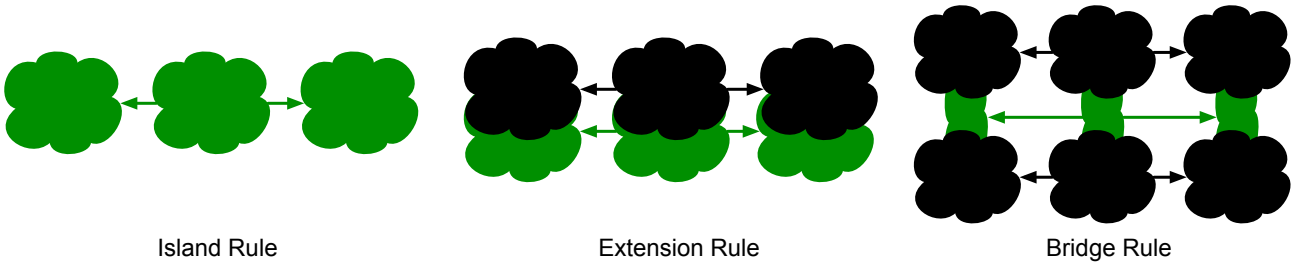


Figure 6: Different kinds of TGG rules: *Islands* (left), *Extensions* (middle), and *Bridges* (right)

Guideline 6 (Prefer islands and bridges to extensions)**.** *When specifying an extension, always ask domain experts if it is acceptable to split the extension into an island and a bridge, as this often improves derived synchronizers. Extensions should only be used if an island would have to change once it is connected via a bridge.*

An extension states that all created elements are dependent on the specified context. In many cases, this introduces unnecessary dependencies between islands. In general, TGG-based synchronization works better, the fewer context dependencies are specified in rules.

Guideline 7 (Formalize information loss via explicit ignore rules)**.** *Most TGG tools attempt to ignore irrelevant elements automatically, as specifying this explicitly with ignore rules can be tedious, especially in the first few iterations, when the TGG covers only a small subset of the source and target metamodels. Explicit ignore rules are nonetheless better than tool-specific heuristics and automatic ignore support, and should be preferred.*

Example: Applying these guidelines to our running example, we now discuss the current TGG (handling package/folder hierarchies) as well as new rules for handling classes and methods, together with their respective documentation. Reflecting on rules `RootPackageToDocuFolder` and `SubPackageToDocuFolder` (cf. Fig 3 and 4) now in the light of our guidelines, we can identify `RootPackageToDocuFolder` as an island, and `SubPackage-ToDocuFolder` as an extension thereof. Can we split the extension into an island and a bridge (G6)? Although subpackages are treated just like root packages, it must be clear from the corresponding documentation file that this is the documentation for a subpackage and not a root package. The heading in the file (`title` in `Root-PackageToDocuFolder`) must, therefore, be adjusted appropriately. Interestingly, if a (sub)package `p` is deleted in the source model, all subpackages of `p` consequently become root packages and must now be (re)translated differently! This also applies to making a former root package a subpackage by connecting it to a super package. In this case, the root package is now a subpackage and must also be (re)translated appropriately. Creating and documenting subpackages in this manner is an example of creating deltas that *cannot* be handled as an island.

Creating a class `c` corresponds to creating an HTML file named `<c.name>` + ".html", this time with a header "EClass" + `<c.name>`. In addition, an empty HTML table to contain all methods of the class should be created in the file. Adding a class `c` to a package `p` corresponds to simply placing the documentation file for `c` into the documentation folder for `p`. This time around, these consistency requirements can be transferred directly to an island and a bridge: Fig. 7 depicts the island `EClassToHTMLFile` for handling the creation of an `EClass` and its documentation file with internal HTML structure. `EClassToHTMLFile` once again refines `HTMLFileAndHeader` to reuse the basic structure of an HTML file. The `body` of the HTML file is extended, however, by a table node `methodTable` and an extra subheader for the table. Attribute constraints are used to ensure that the name and header of the file correspond appropriately to the created class. Specifying this as an island means that classes and their documentation are created in this manner, independent of what package the class is later placed in.

Fig. 7 also depicts the bridge `EClassToFileBridge` used to handle adding classes to packages and their documentation files to the corresponding documentation folder. In this case the "bridge" consists of an edge in each domain. In general, however, an arbitrary number of elements can be created to connect the context clouds as required. The primary advantage of using a bridge here instead of an extension (G6) is that unnecessary dependencies are avoided; a class can be moved to a different package without losing its documentation. Finally, Fig. 7 depicts an ignore rule `IgnoreFileDocu` (no elements are created in the source domain), which states explicitly that adding documentation to an HTML file does not affect the source domain (G7). Note that this handles documentation for both packages and classes as the basic structure of the HTML file is identical.

Fig. 8 depicts an island `DocumentMethod`, a bridge `EOperationTableBridge`, and an ignore rule `Ignore-MethodDocu` used to handle creating methods and their corresponding documentation.

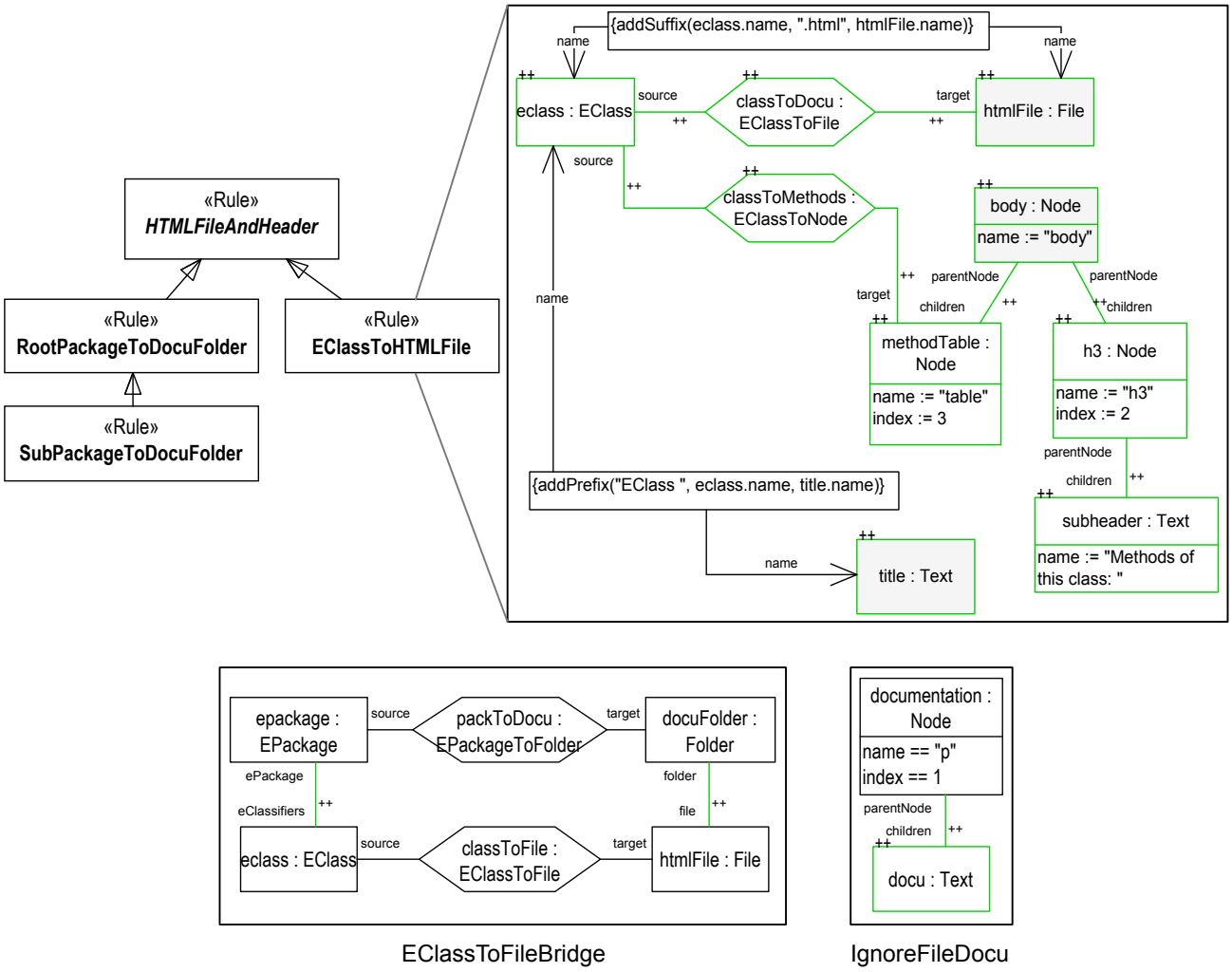


Figure 7: Handling classes and their documentation

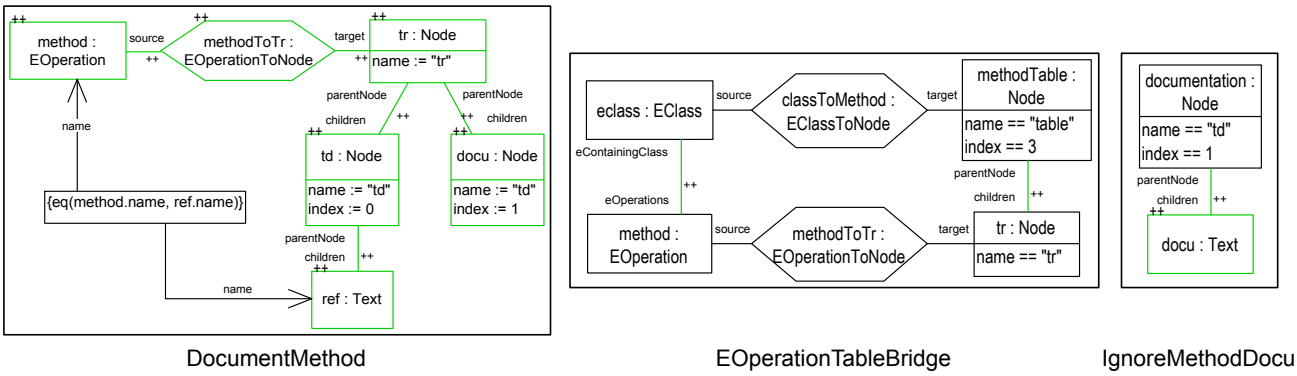


Figure 8: Handling methods and their documentation

Creating a method of a class corresponds to creating a row in an HTML table with two column entries: the first for the name of the method, and the second for its documentation (initially empty). Connecting a method to a class corresponds to adding this row to the method table of the class that was created by `EClassToHTMLFile`. A row in an HTML table is a `tr` element, while column entries are `td` elements. Indices are used in `DocumentMethod` to ensure that the column entries are always in the same order (name before documentation). `EOperationTableBridge` connects a method to a class, and adds its HTML row node to the method table of the class. Analogously to `IgnoreFileDocu`, the ignore rule `IgnoreMethodDocu` states that documenting a method does not affect the source domain. Note that the node `documentation` is constrained to be a column entry.

According to G7, we would actually need to specify ignore rules for all irrelevant concepts in the source domain (cf. Fig. 2) including inheritance relations, attributes, references, and datatypes. This guideline can be relaxed, however, for types that do not occur in *any* TGG rule. It is, for instance, easy to automatically ignore the reference `eSuperTypes` denoting inheritance, but difficult to automatically ignore `Text` documentation nodes (as in `IgnoreMethodDocu`), as the type `Text` does occur, e.g., as `ref:Text` in `DocumentMethod`.

3.4 Profile for hotspots and scalability

Declarative languages such as TGGs certainly have their advantages, but one must reserve some time for profiling and testing for scalability. Depending on the TGG tool and the underlying modelling framework and transformation engine, certain patterns and constructs can be, especially for beginners, surprisingly inefficient.

Guideline 8 (Use a profiler to test regularly for hotspots)**.** *Regularly profiling the synchronization for medium and large sized models is important to avoid bad design decisions early enough in the development process*[4]*.*

To support such a scalability analysis, some TGG tools [11, 23] provide support for generating models (of theoretically arbitrary size) using the TGG specification directly. This should be exploited if available.

Concrete optimization strategies for TGGs are discussed in detail in [15]. The most common user-related optimization is to provide more specific (and in some cases redundant) context elements and attribute constraints within the source and target domains of a rule. This helps the transformation engine eliminate inappropriate rule applications in an early phase, i.e., before checking all (costly) inter-model connections in a rule pattern.

4 TGG-Based Synchronization Algorithms

In this section we try to bring our TGG to life in the synchronization scenario depicted in Fig. 9. Our goal is to impart a high-level intuition for how TGG-based synchronization algorithms work in general. This provides not only a rationale for the guidelines already provided in the previous section, but also an understanding for how arbitrary deltas (and not only the explicitly specified creating deltas) are propagated. How this propagation works arguably depends on the chosen TGG tool, in our case eMoflon whose algorithm is described in [1], and we refer to [12, 16] for a detailed comparison of TGG tools. Nevertheless, the provided intuition is still helpful in understanding the TGG-related consequences of a delta, i.e., which TGG rule applications from former runs must be invalidated or preserved, and which TGG rules must be applied in a new run. In the following, the labels ① – ⑥ are used to refer to certain parts of the synchronization scenario in Fig. 9.

Batch Translation ①, ②: The scenario starts with the creation of a new class diagram ①. To provide a consequent delta-based intuition, this initial step can also be seen as a large source delta consisting of adding all elements in the class diagram. To create this class diagram ①, a root folder `TGGLanguage` containing two classes and two subfolders `compiler` and `precompiler` must be created. Each subfolder also contains a class, and `compiler` contains a subfolder as well. To propagate this source delta ②, TGG-based algorithms compute a sequence of TGG rule applications that would create the resulting source model. A possible sequence in this case is: (1) `RootPackageToDocuFolder` to create `TGGLanguage`, (2) `SubPackageToDocuFolder` applied three times to create `compiler`, `precompiler`, and `compilerfacade`, (3) `EClassToHTMLFile` to create all classes, and (4) `EClassFileBridge` to connect all classes to their containing packages. As TGG rules are specified as *triple* rules, applying this sequence of rules yields not only the expected source model but also a correspondence and a target model. In addition, the resulting triple of source, correspondence, and target models is, by definition, in the specified TGG language and is, therefore, *correct*. The target model is thus the result of forward propagating ② the source delta ①. Although this search for a possible rule application sequence can be implemented naïvely

[4] Although this ultimately depends on the specific metamodels and TGG rules, our current experience is that models of up to about 200 000 elements can be handled reasonably well by TGG-based synchronizers [15].

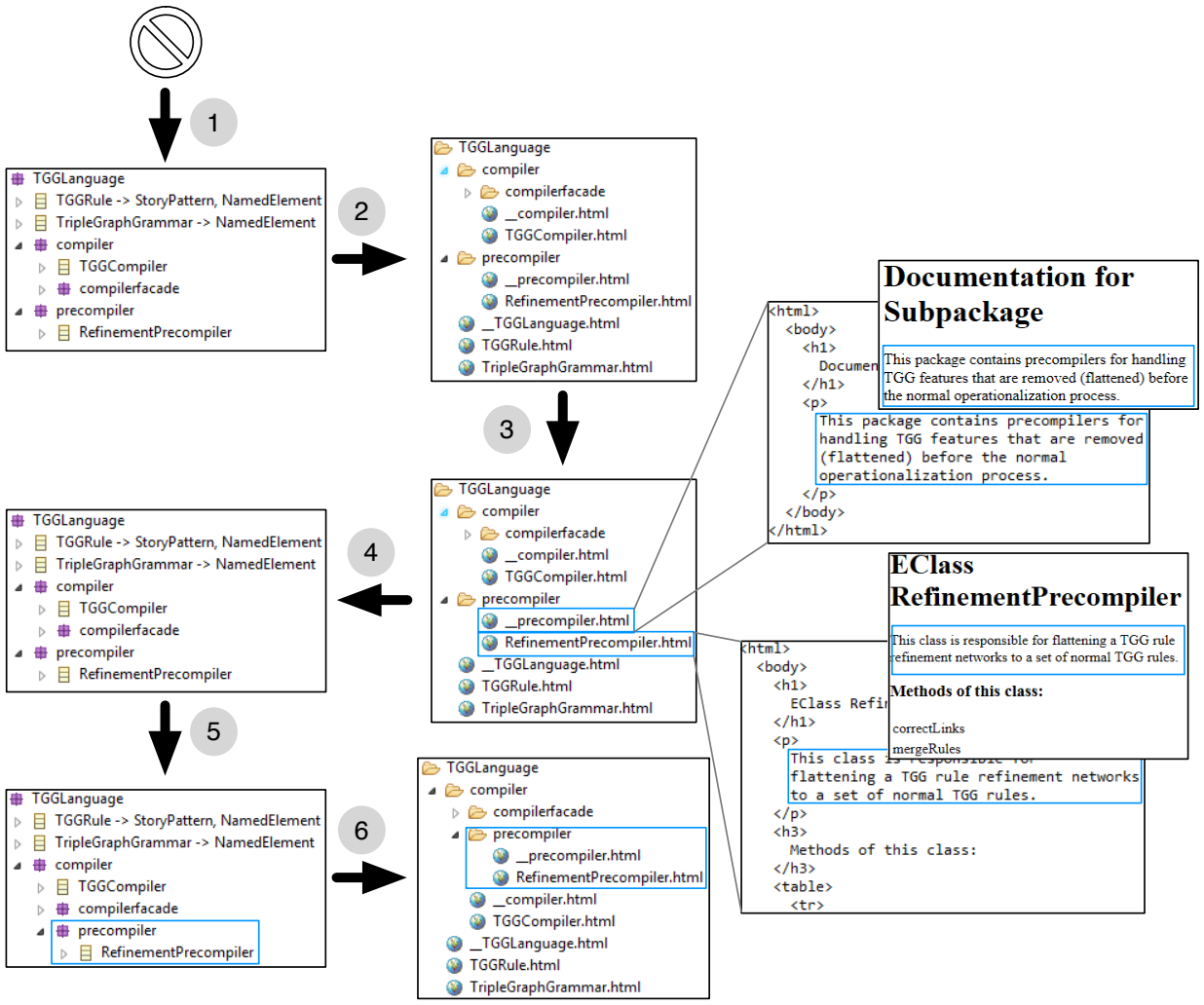


Figure 9: Synchronization scenario with our running example

as a straightforward trial and error procedure with backtracking, this would have exponential runtime and does not scale. All TGG tools we are aware of do not backtrack and instead restrict the class of possible TGGs.

A common restriction is demanding some variant of *confluence*, a well-known property of transition systems, which can be checked for statically using a so-called *critical pair analysis*. The interested reader is referred to, e.g., [2] for details. The naïve understanding of just fitting TGG rules until the correct sequence of rule applications is determined is, however, sufficient for an intuitive understanding of how this works in theory.

Ignoring Changes with Ignore Rules ③, ④: A target delta consisting of two changes is applied in ③. As depicted in Fig. 9, the documentation files `_precompiler.html` and `RefinementPrecompiler.html` for the package `precompiler` and the class `RefinementPrecompiler`, respectively, are edited. In terms of our HTML metamodel, this corresponds to adding a textual node to the corresponding paragraphs (`p` nodes) in the HTML files. A TGG-based synchronizer would propagate ④ this target delta by simply doing nothing, i.e., the source model is not changed at all. The same process is taken as with ②, i.e., a sequence of TGG rules is determined that applies the given target delta. The sequence in this case is: `IgnoreFileDocu` applied twice to create the added text node in both HTML files. As both rules are specified as *ignore* rules, the correspondence and target models are not changed in the process.

Handling Deleted and Added Elements ⑤, ⑥: After discussing two simple cases, we now demonstrate how the choice between extensions and bridges affects the behaviour of a TGG-based synchronizer. Let us consider a more general source delta ⑤, which "moves" the subpackage `precompiler` from `TGGLanguage` to `compiler`.

This is accomplished by (i) deleting the link between `TGGLanguage` and `precompiler`, and (ii) creating a new link between `compiler` and `precompiler`. Propagating this source delta ⑥ thus entails handling a deletion and an addition. The synchronization is, therefore, executed in three phases: a deletion phase, an addition phase, and a translation phase:

```
for all deletions:  revoke all dependent rule applications
for all additions:  revoke all dependent rule applications
translate all revoked and newly added elements
```

To *revoke* means to rollback a rule application, i.e., for forward synchronization, all correspondence and target elements created by the rule application to be revoked are deleted, while all created source elements are considered as revoked and to be (re)translated in the ensuing translation phase.

For every element that has been deleted, all rule applications that require the element as context and thus *depend* on the deleted element become invalid and must be revoked. Note that this must be done transitively.

In general, additions also have to be handled analogously to deletions, i.e., sometimes rule applications must be revoked as a consequence of newly added elements. For our running example, consider *adding* a new root package `eMoflonLanguages` that contains the current root package `TGGLanguage`. This must lead to revoking `TGGLanguage` and re-translating it as a subpackage of the new root package `eMoflonLanguages`!

To explain this further with our synchronization scenario, Fig. 10 depicts a relevant excerpt of the source and target models involved (top left). The deleted link is depicted bold and red with a "`--`" markup for emphasis. A TGG-based synchronizer typically keeps track of the translation by grouping links and objects into *rule applications*. This grouping is depicted visually in Fig. 10 for all rule applications required to create the current triple. For example, the rule application `3:SubPackageToDocuFolder` comprises the deleted link between `TGGLanguage` and `precompiler`, the subpackage `precompiler`, as well as the corresponding target elements: the link between the root folder `TGGLanguage` and subfolder `precompiler`, the folder `precompiler`, and the HTML file `__precompiler.html`. Note that the rule applications also comprise all relevant correspondence elements, which are abstracted from in Fig. 10 to simplify the explanation.

This grouping into rule applications, which can be reconstructed by simulating the creation of a consistent triple from scratch if necessary, is used to determine dependencies between the groups of elements. This is depicted visually in Fig. 10 (top right) as a dependency graph, showing that, for example, the rule application `5:EClassFileBridge` depends on (shown as an arrow) `3:SubPackageToDocuFolder` and `4:EClassToHTMLFile`, as these two rule applications create objects that are required as context in `5:EClassFileBridge`.

Given a source delta and calculated dependencies between rule applications, the deletion phase is carried out by revoking all rule application containing deleted elements, recursively revoking all rule applications that directly or transitively depend on revoked rule applications, and finally removing the deleted source elements from all data structures. This process is depicted in Fig. 10 (top right) showing (with a bold, red outline and "`--`" markup) that `3:SubPackageToDocuFolder` is to be revoked as it contains the deleted source link between `TGGLanguage` and `precompiler`. In this case, the only dependent rule application is `5:EClassFileBridge`, which must also be revoked (bottom right of Fig. 10).

In the translation phase, all revoked source elements are treated as if they were newly added, i.e., they are translated together with all added elements. This is depicted in Fig. 10 (bottom left) showing (bold, green outline and "`++`" markup) the revoked source elements `precompiler` and the link between `precompiler` and `RefinementPrecompiler`, as well as the newly added link between `compiler` and `precompiler` from the source delta. In this state, the same translation strategy as explained for ② and ④ can be applied, however, only for all added and revoked source elements. Due to this process, the documentation added to `RefinementPrecompiler.html` in ③ is retained as its corresponding class is not revoked (Fig. 10, bottom left). As the subpackage `precompiler` is, however, revoked and re-translated, its documentation file `__precompiler.html` is deleted, losing the changes made in ③, and is re-created afresh.

This is a direct consequence of using an extension rule for handling subpackages, but a bridge to connect classes to their parent packages. Similarly, the decision to use a bridge to connect methods to their classes enables, e.g., a *pull-up method* refactoring in the class diagram without having to revoke the method and lose its documentation. One can certainly argue that this behaviour is not always optimal, but it is what can currently be expected from state-of-the-art TGG-based synchronizers, given the choices we made when designing our TGG. Further improving current TGG synchronization algorithms to handle extensions as bridges during change propagation and still guarantee correctness is ongoing research.

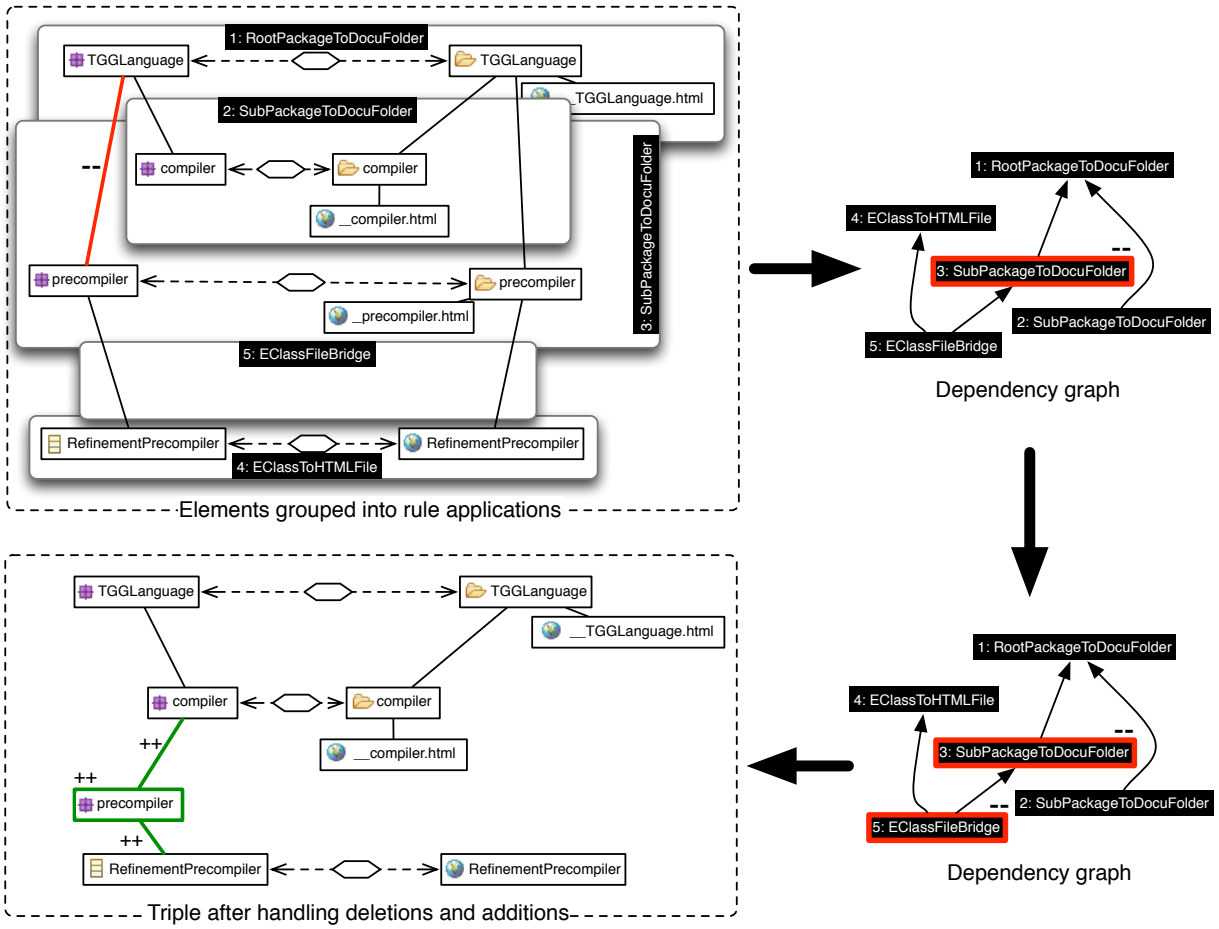


Figure 10: Handling of additions and deletions by a TGG-based synchronization algorithm

5 Related Work

We consider three different groups of related work in the following: (1) previous TGG publications with a focus similar to this work, (2) guidelines for *bx* approaches based on QVT-R which, similar to TGGs, address *bx* in an MDE context, and (3) other approaches to systematically developing model transformation rules in general.

Other approaches to systematically developing a TGG: Kindler and Wagner [13, 22] propose to semi-automatically synthesize TGG rules from exemplary pairs of consistent source and target models. Viewing their iterative synthesizing procedure through our geographical intuition, an island is first constructed from a given pair of consistent models, and is then stepwise deconstructed to smaller islands, bridges, or extensions by removing parts already covered by rules synthesized from former runs. As the procedure largely depends on the complexity of the transformation as well as the representativeness and conciseness of the provided examples, the authors suggest finalizing the process with manual modifications, for which our guidelines can be used.

Engineering guidelines are presented in [15] based on experience with profiling and optimizing TGGs. In contrast to this paper, the topics handled in [15] are mostly scalability-oriented and address, in many cases, primarily TGG tool developers rather than end users.

Guidelines for *bx* with QVT-R: Similar to TGGs, QVT-Relations (QVT-R) addresses *bx* in an MDE context. The standard QVT-R reference [17] already supplies examples demonstrating the usage of different language constructs. The standardized textual concrete syntax facilitates the distribution of such best practices across different QVT-R tools, whereas TGG tools currently suffer from interoperability issues as different metamodels are used for representing graphs, rules, and correspondences. Considering the chronological order of formal and practical contributions for TGGs and QVT-R, one can observe that while we now strive to impart a practical intuition and guidelines to complement existing formal work on TGGs, recent papers on QVT-R [4, 9, 20] strive to formalize concepts for an existing intuition.

Approaches to designing model transformation rules in general: Most of the work to facilitate transformation rule development focuses on semi-automatic usage of exemplary model pairs. While not necessarily focusing on *bx*, such approaches (e.g., [21, 24]) often require explicit mappings on the instance level, which closely resemble the correspondences in TGG rules. Further related contributions are those based on *design patterns* for model transformation [7, 10]. Although such design languages help to share solution strategies based on a common representation, our experience (especially with TGGs and QVT-R) is that the concrete choice of transformation language has a substantial impact on the proper way of thinking about a notion of consistency. In case of TGGs, for example, one must refrain from planning with control flow structures or (recursive) explicit rule invocations; features that are not necessarily excluded from other (*bx*) languages or general design pattern languages. Finally, this paper was inspired by the work of Zambon and Rensink in [25], where they demonstrate best practices for the transformation tool GROOVE using the N-Queens problem.

6 Conclusion and Future Work

In this paper, we have presented not only a basic introduction to TGGs, but also a process and a set of guidelines to support the systematic development of a TGG from a clear, but unformalised understanding of a *bx*.

We have, however, only been able to handle the basics and leave guidelines for advanced language features and techniques to future work including: negative application conditions, multi-amalgamation, user-defined attribute manipulation, test generation, and how best to specify the correspondence metamodel.

Acknowledgements. The first author of this paper received partial funding from the European Union's Seventh Framework Program (FP7/2007-2013) for CRYSTAL-Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830 and from Vinnova under DIARIENR 2012-04304.

References

[1] Anthony Anjorin. *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*. Phd thesis, Technische Universität Darmstadt, 2014.

[2] Anthony Anjorin, Erhan Leblebici, Andy Schürr, and Gabriele Taentzer. A Static Analysis of Non-Confluent Triple Graph Grammars for Efficient Model Transformation. In Holger Giese and Barbara König, editors, *ICGT 2014*, volume 8571 of *LNCS*, pages 130–145. Springer, 2014.

[3] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In Stefania Gnesi and Arend Rensink, editors, *FASE 2014*, volume 8411 of *LNCS*, pages 340–354. Springer, 2014.

[4] Julian C. Bradfield and Perdita Stevens. Recursive Checkonly QVT-R Transformations with General when and where Clauses via the Modal Mu Calculus. In Juan de Lara and Andrea Zisman, editors, *FASE 2012*, volume 7212 of *LNCS*, pages 194–208. Springer, 2012.

[5] James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a Repository of Bx Examples. In K. Selccuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *BX 2014*, volume 1133 of *CEUR Workshop Proc.*, pages 87–91, 2014.

[6] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[7] Hüseyin Ergin and Eugene Syriani. Towards a Language for Graph-Based Model Transformation Design Patterns. In Davide Di Ruscio and Dániel Varró, editors, *ICMT 2014*, volume 8568 of *LNCS*, pages 91–105. Springer, 2014.

[8] Joel Greenyer and Jan Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE 2011*, volume 7233 of *LNCS*, pages 222–237. Springer, 2012.

[9] Esther Guerra and Juan de Lara. An Algebraic Semantics for QVT-Relations Check-only Transformations. *Fundamentae Informatica*, 114(1):73–101, 2012.

[10] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. Engineering model transformations with transML. *SoSym*, 12(3):555–577, 2013.

[11] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE 2011*, volume 7233 of *LNCS*, pages 238–253. Springer, 2011.

[12] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A Survey of Triple Graph Grammar Tools. In Perdita Stevens and James Terwilliger, editors, *BX 2013*, volume 57 of *ECEASST*. EASST, 2013.

[13] Ekkart Kindler and Robert Wagner. Triple Graph Grammars : Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[14] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC-FSE 2007*, pages 285–294. ACM, 2007.

[15] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A Catalogue of Optimization Techniques for Triple Graph Grammars. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014*, volume 225 of *LNI*, pages 225–240. Gesellschaft für Informatik, 2014.

[16] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. In Frank Hermann and Stefan Sauer, editors, *GT-VMT 2014*, volume 67 of *ECEASST*. EASST, 2014.

[17] OMG. MOF2.0 query/view/transformation (QVT) version 1.2. OMG document formal/2015-02-01, 2015. Available from www.omg.org.

[18] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG 1994*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[19] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE 2007*, volume 5235 of *LNCS*, pages 408–424. Springer, 2008.

[20] Perdita Stevens. A Simple Game-Theoretic Approach to Checkonly QVT Relations. *SoSym*, 12(1):175–199, 2013.

[21] Dániel Varró. Model Transformation by Example. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS 2006*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.

[22] Robert Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, 2009.

[23] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In Davide Di Ruscio and Dániel Varró, editors, *ICMT 2014*, volume 8568 of *LNCS*, pages 1–16. Springer, 2014.

[24] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By Example. In *HICSS-40 2007*, page 285. IEEE, 2007.

[25] Eduardo Zambon and Arend Rensink. Solving the N-Queens Problem with GROOVE - Towards a Compendium of Best Practices. In *GTVMT 2014*, volume 67 of *ECEASST*. EASST, 2014.